# 目錄

# Scala 初学指南

## 这本书是什么

本书是 The Neophyte's Guide to Scala 的中文翻译。 *The Neophyte's Guide to Scala* 是 Daniel Westheide 写的一系列有关 Scala 的文章。

原作者在 leanpub 中将此系列文章打包成书，如果你觉得此书对你有帮助， 请到 这里 给予原作者支持！

## 为什么会有这本书

在读书和学习过程中，个人一直很难坚持做好一件事情。 对于专业知识，凡不明白的，我都想一探到底， 但由于自身的知识缺陷太大，很多东西到最后都无法坚持下来，很容易产生挫败感。 遂借此机会，试图克服这一缺点。

目前已经花了整整一个月的时间去翻译和校对，基本算是完成，不愧初心。 当然，凡事都不是尽善尽美的，翻译中必然存在理解偏差和用词不当的地方。 欢迎任何 Issues 和 Pull Requests ！

# 介绍

2012年秋天，超过五万人注册了 Martin Odersky 先生在 Coursera 上开设的 Functional Programming Principles in Scala 课程。 这是一个巨大的数字。 他们可能是第一次接触 Scala、函数式编程。 2013年，这个课程又开始了，并将更多的学生和开发者带入了 Scala 和函数式编程的世界。

如果你正在看这篇文章，很可能你也是其中之一，或者通过其他方式已经学习 Scala 了。 不管怎样，如果你对探索这门优美的语言感到兴奋，但又不知道该如何去学，那这本书就是为你准备的。

尽管 Coursera 上的这门课程已经提供了很多材料来介绍 Scala， 但其时间有限，很难把所有东西都解释清楚， 对于初学者的你，Scala 的一些特性看起来就像魔法一样。 可能你知道如何使用它们，但无法完全掌握其背后的原理， 更重要的是，你无法了解为什么这样做就是对的。

自从这门课程的第一次开设，我就开始 撰写一系列博客， 意在把事情理清楚，移除初学者心中的问号。 这份电子书就基于这系列博客。 鉴于超多人都给出了正面评价，我决定把所有文章编译成书。

在这本书里，我会解释 Scala 语言的一些特性，一些我曾经遇到过麻烦的特性。 之前大部分时候，我找不到对这些特性的好的解释，只能摸石头过河。 为了不让读者步我的后尘，我会在写作中给出这些特性的惯例用法。

介绍的已经差不多了。 在开始这本书之前，读者要知道，虽然并不要求参与过 Coursera 上的那门课程， 但是之前上过 Coursera 的 Scala 课程，会有利于本书的阅读， 我时不时也会引用课程上的一些知识点。

# 提取器

在 Coursera 上，想必你遇到过一个非常强大的语言特性：模式匹配 。 它可以解绑一个给定的数据结构。 这不是 Scala 所特有的，在其他出色的语言中，如 Haskell、Erlang，模式匹配也扮演着重要的角色。

模式匹配可以解构各种数据结构，包括 列表 、 流 ，以及 样例类 。 但只有这些数据结构才能被解构吗，还是可以用某种方式扩展其使用范围？ 而且，它实际是怎么工作的？ 是不是有什么魔法在里面，得以写些类似下面的代码？

```
case class User(firstName: String, lastName: String, score: Int)

def advance(xs: List[User]) = xs match {
  case User(_, _, score1) :: User(_, _, score2) :: _ => score1
- score2
  case _ => 0
}
```

事实证明没有什么魔法，这都归功于提取器 。

提取器使用最为广泛的使用有着与 构造器 相反的效果： 构造器从给定的参数列表创建一个对象， 而提取器却是从传递给它的对象中提取出构造该对象的参数。 Scala 标准库包含了一些预定义的提取器，我们会大致的了解一下它们。

样例类非常特殊，Scala会自动为其创建一个 伴生对象 ： 一个包含了 `apply` 和 `unapply` 方法的 单例对象 。 `apply` 方法用来创建样例类的实例，而 `unapply` 需要被伴生对象实现，以使其成为提取器。

## 第一个提取器

`unapply` 方法可能不止有一种方法签名， 不过，我们从只有最简单的开始，毕竟使用更广泛的还是只有一种方法签名的 `unapply` 。 假设要创建了一个 `User` 特质，有两个类继承自它，并且包含一个字段：

```
trait User {
  def name: String
}
class FreeUser(val name: String) extends User
class PremiumUser(val name: String) extends User
```

我们想在各自的伴生对象中为 `FreeUser` 和 `PremiumUser` 类实现提取器，就像 Scala 为样例类所做的一样。如果想让样例类只支持从给定对象中提取单个参数，那 `unapply` 方法的签名看起来应该是这个样子：

```
  def unapply(object: S): Option[T]
```

这个方法接受一个类型为 `S` 的对象，返回类型 `T` 的 `Option` ， `T` 就是要提取的参数类型。

> 在Scala中， `Option` 是 `null` 值的安全替代。以后会有一个单独的章节来讲述它，不过现在，只需要知道， `unapply` 方法要么返回 `Some[T]` （如果它能成功提取出参数），要么返回 `None` ， `None` 表示参数不能被 `unapply` 具体实现中的任一提取规则所提取出。

下面的代码是我们的提取器：

```
trait User {
  def name: String
}
class FreeUser(val name: String) extends User
class PremiumUser(val name: String) extends User
object FreeUser {
  def unapply(user: FreeUser): Option[String] = Some(user.name)
}
object PremiumUser {
  def unapply(user: PremiumUser): Option[String] = Some(user.nam
e)
}
```

现在，可以在REPL中使用它：

```
scala> FreeUser.unapply(new FreeUser("Daniel"))
res0: Option[String] = Some(Daniel)
```

如果调用返回的结果是 `Some[T]` ，说明提取模式匹配成功，如果是 `None` ，说明模式不匹配。

一般不会直接调用它，因为用于提取器模式时，Scala 会隐式的调用提取器的 `unapply` 方法。

```
  val user: User = new PremiumUser("Daniel")
  user match {
    case FreeUser(name) => "Hello" + name
    case PremiumUser(name) => "Welcome back, dear" + name
  }
```

你会发现，两个提取器绝不会都返回 `None` 。 这个例子展示的提取器要比之前所见的更有意义。 如果你有一个类型不确定的对象，你可以同时检查其类型并解构。

这个例子里， `FreeUser` 模式并不会匹配，因为它接受的类型和我们传递给它的不一样。 这样一来， `user` 对象就会被传递给第二个模式，也就是 `PremiumUser` 伴生对象的 `unapply` 方法。 而这个模式会匹配成功，从而返回值就被绑定到 `name` 参数上。

在接下来的文章里，我们会看到一个并不总是返回 `Some[T]` 的提取器的例子。

## 提取多个值

现在，假设类有多个字段：

```
trait User {
  def name: String
  def score: Int
}
class FreeUser(
  val name: String,
  val score: Int,
  val upgradeProbability: Double
) extends User
class PremiumUser(
  val name: String,
  val score: Int
) extends User
```

如果提取器想解构出多个参数，那它的 `unapply` 方法应该有这样的签名：

```
def unapply(object: S): Option[(T1, ..., T2)]
```

这个方法接受类型为 `S` 的对象，返回类型参数为 `TupleN` 的 `Option` 实例，`TupleN` 中的 `N` 是要提取的参数个数。

修改类之后，提取器也要做相应的修改：

```
trait User {
  def name: String
  def score: Int
}
class FreeUser(
  val name: String,
  val score: Int,
  val upgradeProbability: Double
) extends User
class PremiumUser(
  val name: String,
  val score: Int
) extends User
object FreeUser {
  def unapply(user: FreeUser): Option[(String, Int, Double)] =
    Some((user.name, user.score, user.upgradeProbability))
}
object PremiumUser {
  def unapply(user: PremiumUser): Option[(String, Int)] =
    Some((user.name, user.score))
}
```

现在可以拿它来做模式匹配了：

```
val user: User = new FreeUser("Daniel", 3000, 0.7d)
user match {
  case FreeUser(name, _, p) =>
    if (p > 0.75) "$name, what can we do for you today?"
    else "Hello $name"
  case PremiumUser(name, _) =>
    "Welcome back, dear $name"
}
```

## 布尔提取器

有些时候，进行模式匹配并不是为了提取参数，而是为了检查其是否匹配。 这种情况下，第三种 `unapply` 方法签名（也是最后一种）就有用了， 这个方法接受 `S` 类型的对象，返回一个布尔值：

```
def unapply(object: S): Boolean
```

使用的时候，如果这个提取器返回 `true` ，模式会匹配成功， 否则，Scala 会尝试拿 `object` 匹配下一个模式。

上一个例子存在一些逻辑代码，用来检查一个免费用户有没有可能被说服去升级他的账户。 其实可以把这个逻辑放在一个单独的提取器中：

```
object premiumCandidate {
  def unapply(user: FreeUser): Boolean = user.upgradeProbability
 > 0.75
}
```

你会发现，提取器不一定非要在这个类的伴生对象中定义。 正如其定义一样，这个提取器的使用方法也很简单：

```
val user: User = new FreeUser("Daniel", 2500, 0.8d)
user match {
  case freeUser @ premiumCandidate() => initiateSpamProgram(free
User)
  case _ => sendRegularNewsletter(user)
}
```

使用的时候，只需要把一个空的参数列表传递给提取器，因为它并不真的需要提取数据，自然也没必要绑定变量。

这个例子有一个看起来比较奇怪的地方： 我假设存在一个空想的 `initiateSpamProgram` 函数，其接受一个 `FreeUser` 对象作为参数。 模式可以与任何一种 `User` 类型的实例进行匹配，但 `initiateSpamProgram` 不行， 只有将实例强制转换为 `FreeUser` 类型， `initiateSpamProgram` 才能接受。

因为如此，Scala 的模式匹配也允许将提取器匹配成功的实例绑定到一个变量上，这个变量有着与提取器所接受的对象相同的类型。这通过 `@` 操作符实现。`premiumCandidate` 接受 `FreeUser` 对象，因此，变量 `freeUser` 的类型也就是 `FreeUser` 。

布尔提取器的使用并没有那么频繁（就我自己的情况来说），但知道它存在也是很好的， 或迟或早，你会遇到一个使用布尔提取器的场景。

## 中缀表达方式

解构列表、流的方法与创建它们的方法类似，都是使用 cons 操作符： `::` 、 `#::` ，比如：

```
val xs = 58 #:: 43 #:: 93 #:: Stream.empty
xs match {
  case first #:: second #:: _ => first - second
  case _ => -1
}
```

你可能会对这种做法产生困惑。 除了我们已经见过的提取器用法，Scala 还允许以中缀方式来使用提取器。 所以，我们可以写成 `e(p1, p2)` ，也可以写成 `p1 e p2` ， 其中 `e` 是提取器， `p1` 、 `p2` 是要提取的参数。

同样，中缀操作方式的 `head #:: tail` 可以被写成 `#::(head, tail)` ， 提取器 `PremiumUser` 可以这样使用： `name PremiumUser score` 。 当然，这样做并没有什么实践意义。 一般来说，只有当一个提取器看起来真的像操作符，才推荐以中缀操作方式来使用它。 所以，列表和流的 `cons` 操作符一般使用中缀表达，而 `PreimumUser` 则不用。

## 进一步看流提取器

尽管 `#::` 提取器在模式匹配中的使用并没有什么特殊的， 但是，为了更好的理解上面的代码，还是进一步来分析一下。 而且，这是一个很好的例子，根据要匹配的数据结构的状态，提取器很可能返回 `None` 。

如下是 *Scala 2.9.2* 源代码中完整的 `#::` 提取器代码：

```
object #:: {
  def unapply[A](xs: Stream[A]): Option[(A, Stream[A]) =
    if (xs.isEmpty) None
    else Some((xs.head, xs.tail))
}
```

如果给定的流是空的，提取器就直接返回 `None` 。因此， `case head #:: tail` 就不会匹配任何空的流。否则，就会返回一个 `Tuple2` ，其第一个元素是流的头，第二个元素是流的尾，尾本身又是一个流。这样， `case head #:: tail` 就会匹配有一个或多个元素的流。如果只有一个元素， `tail` 就会被绑定成空流。

为了理解流提取器是怎么在模式匹配中工作的，重写上面的例子，把它从中缀写法转成普通的提取器模式写法：

```
val xs = 58 #:: 43 #:: 93 #:: Stream.empty
xs match {
  case #::(first, #::(second, _)) => first - second
  case _ => -1
}
```

首先为传递给模式匹配的初始流 `xs` 调用提取器。由于提取器返回 `Some(xs.head, xs.tail)` ，从而 `first` 会绑定成 58， `xs` 的尾会继续传递给提取器，提取器再一次被调用，返回首和尾， `second` 就被绑定成 `43` ，而尾就绑定到通配符 `_` ，被直接扔掉了。

## 使用提取器

那到底该在什么时候使用、怎么使用自定义的提取器呢？尤其考虑到，使用样例类就能自动获得可用的提取器。

一些人指出，使用样例类、对样例类进行模式匹配打破了封装，耦合了匹配数据和其具体实现的方式，这种批评通常是从面向对象的角度出发的。如果想用 Scala 进行函数式编程，将样例类当作只包含纯数据（不包含行为）的 代数数据类型 ，那它非常适合。

通常，只有当从无法掌控的类型中提取数据，或者是需要其他进行模式匹配的方法时，才需要实现自己的提取器。

> 提取器的一种常见用法是从字符串中提取出有意义的值， 作为练习，想一想如何实现 `URLExtractor` 以匹配代表 URL 的字符串。

## 小结

在这本书的第一章中，我们学习了 Scala 模式匹配背后的提取器， 学会了如何实现自己的提取器，及其在模式中的使用是如何和实现联系在一起的。 但是这并不是提取器的全部，下一章，将会学习如何实现可提取可变个数参数的提取器。

# 序列提取

上一章讲述了如何实现自定义的提取器以及如何在模式匹配中使用它们， 但是只讨论了如何从给定的数据结构中分解固定数目的参数。 对某种数据结构来说，Scala 提供了提取任意多个参数的模式匹配方法。

比如，你可以匹配只有两个、或者只有三个元素的列表：

```
val xs = 3 :: 6 :: 12 :: Nil
xs match {
 case List(a, b) => a * b
 case List(a, b, c) => a + b + c
 case _ => 0
}
```

除此之外，也可以使用通配符 `_*` 匹配长度不确定的列表：

```
val xs = 3 :: 6 :: 12 :: 24 :: Nil
xs match {
 case List(a, b, _*) => a * b
 case _ => 0
}
```

这个例子中，第一个模式成功匹配，把 `xs` 的前两个元素分别绑定到 `a` 、 `b` ， 而剩余的列表，无论其还有多少个元素，都直接被忽略掉。

显然，这种模式的提取器是无法通过上一章介绍的方法来实现的。 需要一种特殊的方法，来使得一个提取器可以接受某一类型的对象，将其解构成列表， 且这个列表的长度在编译期是不确定的。

`unapplySeq` 就是用来做这件事情的，下面的代码是其可能的方法签名：

```
 def unapplySeq(object: S): Option[Seq[T]]
```

这个方法接受类型 `S` 的对象，返回一个类型参数为 `Seq[T]` 的 `Option` 。

## 例子：提取给定的名字

现在我们举一个例子来展示如何使用这种提取器。

假设有一个应用，其某处代码接收了一个表示人名且类型为 `String` 的参数，这个字符串可能包含了这个人的第二个甚至是第三个名字（如果这个人不止有一个名字）。比如说，`Daniel` 、 `Catherina Johanna` 、 `Matthew John Michael` 。而我们想做的是，从这个字符串中提取出单个的名字。

下面的代码是一个用 `unapplySeq` 方法实现的提取器：

```
object GivenNames {
  def unapplySeq(name: String): Option[Seq[String]] = {
    val names = name.trim.split(" ")
    if (name.forall(_.isEmpty)) None
    else Some(names)
  }
}
```

给定一个含有一个或多个名字的字符串，这个提取器会将其解构成一个列表。如果字符串不包含有任何名字，提取器会返回 `None` ，提取器所在的那个模式就匹配失败。

下面对提取器进行测试：

```
  def greetWithFirstName(name: String) = name match {
    case GivenNames(firstName, _*) => s"Good morning, $firstname
!"
    case _ => "Welcome! Please make sure to fill in your name!"
  }
```

`greetWithFirstName("Daniel")` 会返回 "Good morning, Daniel!"，而 `greetWithFirstName("Catherina Johanna")` 会返回 "Good morning, Catherina!"。

## 固定和可变的参数提取

有些时候，需要提取出至少多少个值， 这样，在编译期，就知道必须要提取出几个值出来，再外加一个可选的序列，用来保存不确定的那一部分。

在我们的例子中，假设输入的字符串包含了一个人完整的姓名，而不仅仅是名字。比如字符串可能是"John Doe"、"Catherina Johanna Peterson"，其中，"Doe"、"Peterson"是姓，"John"、"Catherina"、"Johanna"是名。 我们想做的是匹配这样的字符串，把姓绑定到第一个变量， 把第一个名字绑定到第二个变量，第三个变量存放剩下的任意个名字。

稍微修改 `unapplySeq` 方法就可以解决上述问题：

```
def unapplySeq(object: S): Option[(T1, .., Tn-1, Seq[T])]
```

`unapplySeq` 返回的同样是 `Option[TupleN]` ，只不过，其最后一个元素是一个 `Seq[T]` 。 这个方法签名看起来应该很熟悉，它和之前的一个 `unapply` 签名类似。

下列代码是利用这个方法生成的提取器：

```
object Names {
  def unapplySeq(name: String): Option[(String, String, Seq[Stri
ng])] = {
    val names = name.trim.split(" ")
    if (names.size < 2) None
    else Some((names.last, names.head, names.drop(1).dropRight(1
)))
  }
}
```

仔细看看其返回值，及其构造 `Some` 的方式。 代码返回一个类型参数为 `Tuple3` 的 `Option` ，这个元组包含了姓、名、以及由剩余的名字构成的序列。

如果这个提取器用在一个模式中，那只有当给定的字符串至少含有姓和名时，模式才匹配成功。

下面用这个提取器重写 `greeting` 方法：

```
def greet(fullName: String) = fullName match {
  case Names(lastName, firstName, _*) =>
    s"Good morning, $firstName $lastName!"
  case _ =>
    "Welcome! Please make sure to fill in your name!"
}
```

你可以在 REPL 中或者 worksheet 上试试这些代码。

## 小结

这一章里，我们学会了怎样去实现和使用返回不定长度值序列的提取器。 提取器是一个相当强大的工具，你可以灵活的重用它们，从而提供一种有效的方法来扩展要匹配的模式。

下一章，我会给出模式匹配在 Scala 中的不同用法（现在所见到的只是冰山一角）。

# 无处不在的模式

前两章花费了相当多的时间去解释下面这两件事情：

1. 用模式解构对象是怎么一回事。
2. 如何构造自己的提取器。

现在是时候去了解模式更多的用法了。

## 模式匹配表达式

模式可能出现的一个地方就是 模式匹配表达式*(pattern matching expression)*： 一个表达式 `e` ，后面跟着关键字 `match` 以及一个代码块，这个代码块包含了一些匹配样例； 而样例又包含了 `case` 关键字、模式、可选的 守卫分句*(guard clause)* ，以及最右边的代码块； 如果模式匹配成功，这个代码块就会执行。 写成代码，看起来会是下面这种样子：

```
e match {
  case Pattern1 => block1
  case Pattern2 if-clause => block2
  ...
}
```

下面是一个更具体的例子：

```
case class Player(name: String, score: Int)
def printMessage(player: Player) = player match {
  case Player(_, score) if score > 100000 =>
    println("Get a job, dude!")
  case Player(name, _) =>
    println("Hey, $name, nice to see you again!")
}
```

`printMessage` 的返回值类型是 `Unit` ，其唯一目的是执行一个副作用，即打印一条信息。 要记住你不一定非要使用模式匹配，因为你也可以使用像 Java 语言中的 switch 语句。

但这里使用的模式匹配表达式之所以叫 模式匹配表达式 是有原因的： 其返回值是由第一个匹配的模式中的代码块决定的。

使用它通常是好的，因为它允许你解耦两个并不真正属于彼此的东西，也使得你的代码更易于测试。 可把上面的例子重写成下面这样：

```
case class Player(name: String, score: Int)
def message(player: Player) = player match {
  case Player(_, score) if score > 100000 =>
    "Get a job, dude!"
  case Player(name, _) =>
    "Hey, $name, nice to see you again!"
}
def printMessage(player: Player) = println(message(player))
```

现在，独立出一个返回值是 `String` 类型的 `message` 函数， 它是一个纯函数，没有任何副作用，返回模式匹配表达式的结果， 你可以将其保存为值，或者赋值给一个变量。

# 值定义中的模式

模式还可能出现值定义的左边。 （以及变量定义，本书中变量的使用并不多，因为我偏向于使用函数式风格的Scala代码）

假设有一个方法，返回当前的球员，我们可以模拟这个方法，让它始终返回同一个球员：

```
def currentPlayer(): Player = Player("Daniel", 3500)
```

通常的值定义如下所示：

```
val player = currentPlayer()
doSomethingWithName(player.name)
```

如果你知道 Python，你可能会了解一个称为 序列解包*(sequence unpacking)* 的功能， 它允许在值定义（或者变量定义）的左侧使用模式。 你可以用类似的风格编写你的 Scala 代码：改变我们的代码，在将球员赋值给左侧变量的同时去解构它：

```
val Player(name, _) = currentPlayer()
doSomethingWithName(name)
```

你可以用任何模式来做这件事情，但得确保模式总能够匹配，否则，代码会在运行时出错。下面的代码就是有问题的： `scores` 方法返回球员得分的列表。为了说明问题，代码中只是返回一个空的列表。

```
def scores: List[Int] = List()
val best :: rest = scores
println("The score of our champion is " + best)
```

运行的时候，就会出现 `MatchError` 。（好像我们的游戏不是那么成功，毕竟没有任何得分）

一种安全且非常方便的使用方式是只解构那些在编译期就知道类型的样例类。此外，以这种方式来使用元组，代码可读性会更强。假设有一个函数，返回一个包含球员名字及其得分的元组，而不是先前定义的 `Player` ：

```
def gameResult(): (String, Int) = ("Daniel", 3500)
```

访问元组字段的代码给人感觉总是很怪异：

```
val result = gameResult()
println(result._1 + ": " + result._2)
```

这样，在赋值的同时去解构它是非常安全的，因为我们知道它类型是 `Tuple2` ：

```
val (name, score) = gameResult()
println(name + ": " + score)
```

这就好看多了，不是吗？

## for 语句中的模式

模式在 for 语句中也非常重要。 所有能在值定义的左侧使用的模式都适用于 for 语句的值定义。 因此，如果我们有一个球员得分集，想确定谁能进名人堂（得分超过一定上限），用 for 语句就可以解决：

```
def gameResults(): Seq[(String, Int)] =
  ("Daniel", 3500) :: ("Melissa", 13000) :: ("John", 7000) :: Nil

def hallOfFame = for {
    result <- gameResults()
    (name, score) = result
    if (score > 5000)
  } yield name
```

结果是 `List("Melissa", "John")` ，因为第一个球员得分没超过 5000。

上面的代码还可以写的更简单，for 语句中，生成器的左侧也可以是模式。 从而，可以直接在左则把想要的值解构出来：

```
def hallOfFame = for {
    (name, score) <- gameResults()
    if (score > 5000)
  } yield name
```

模式 `(name, score)` 总会匹配成功， 如果没有守卫语句 `if (score > 5000)` ， for 语句就相当于直接将元组映射到球员名字，不会进行过滤。

不过你要知道，生成器左侧的模式也可以用来过滤。 如果左侧的模式匹配失败，那相关的元素就会被直接过滤掉。

为了说明这种情况，假设有一序列的序列，我们想返回所有非空序列的元素个数。 这就需要过滤掉所有的空列表，然后再返回剩下列表的元素个数。 下面是一个解决方案：

```
val lists = List(1, 2, 3) :: List.empty :: List(5, 3) :: Nil
for {
  list @ head :: _ <- lists
} yield list.size
```

上面例子中，左侧的模式不匹配空列表。 这不会抛出 `MatchError` ，但对应的空列表会被丢掉，因此得到的结果是 `List(3, 2)` 。

模式和 for 语句是一个很自然、很强大的结合。 用 Scala 工作一段时间后，你会发现经常需要它。

## 小结

这一章讲述了模式的多种使用方式。 除此之外，模式还可以用于定义匿名函数，如果你试过用 `catch` 块处理 Scala 中的异常，那你就见过模式的这个用法， 下一章会详细描述。

# 模式匹配与匿名函数

上一章总结了模式在 Scala 中的几种用法，最后提到了匿名函数。 这一章，我们具体的去学习如何在匿名函数中使用模式。

如果你参与过 Coursera 上的 那门 Scala 课程 ， 或者写过 Scala 代码，那很可能你已经熟悉匿名函数。 比如说，将一组歌名转换成小写格式，你可能会定义一个匿名函数传递给 `map` 方法：

```
val songTitles = List("The White Hare", "Childe the Hunter", "Take no Rogues")
songTitles.map(t => t.toLowerCase)
```

或者，利用 Scala 的 占位符语法*(placeholder syntax)* 得到更加简短的代码：

```
songTitles.map(_.toLowerCase)
```

目前为止，一切都很顺利。 不过，让我们来看一个稍微有些区别的例子： 假设有一个由二元组组成的序列，每个元组包含一个单词，以及对应的词频， 我们的目标就是去除词频太高或者太低的单词，只保留中间地带的。 需要写出这样一个函数：

```
wordsWithoutOutliers(wordFrequencies: Seq[(String, Int)]): Seq[String]
```

一个很直观的解决方案是使用 `filter` 和 `map` 函数：

```
val wordFrequencies = ("habitual", 6) :: ("and", 56) :: ("consuetudinary", 2) ::
  ("additionally", 27) :: ("homely", 5) :: ("society", 13) :: Nil

def wordsWithoutOutliers(wordFrequencies: Seq[(String, Int)]): Seq[String] =
  wordFrequencies.filter(wf => wf._2 > 3 && wf._2 < 25).map(_._1)
wordsWithoutOutliers(wordFrequencies) // List("habitual", "homely", "society")
```

这个解法有几个问题。 首先，访问元组字段的代码不好看，如果我们可以直接解构出字段，那代码可能更加美观和可读。

幸好，Scala 提供了另外一种写匿名函数的方式：模式匹配形式的匿名函数， 它是由一系列模式匹配样例组成的，正如模式匹配表达式那样，不过没有 `match` 。下面是重写后的代码：

```
def wordsWithoutOutliers(wordFrequencies: Seq[(String, Int)]): Seq[String] =
 wordFrequencies.filter { case (_, f) => f > 3 && f < 25 } map { case (w, _) => w }
```

在两个匿名函数里，我们只使用了一个匹配案例，因为我们知道这个样例总是会匹配成功， 要解构的数据类型在编译期就确定了，没有会出错的可能。 这是模式匹配型匿名函数的一个非常常见的用法。

如果把这些匿名函数赋给其他值，你也会看到它们有着正确的类型：

```
val predicate: (String, Int) => Boolean = { case (_, f) => f > 3 && f < 25 }
val transformFn: (String, Int) => String = { case (w, _) => w }
```

> 不过要注意，必须显示的声明值的类型，因为 Scala 编译器无法从匿名函数中推导出其类型。

当然，也可以定义一系列更加复杂的的匹配案例。 但是你必须的确保对于每一个可能的输入，都会有一个样例能够匹配成功， 不然，运行时会抛出 `MatchError` 。

## 偏函数

有时候可能会定义一个只处理特定输入的函数。 这样的一种函数能帮我们解决 `wordsWithoutOutliers` 中的另外一个问题： 在 `wordsWithoutOutliers` 中，我们首先过滤给定的序列，然后对剩下的元素进行映射， 这种处理方式需要遍历序列两次。 如果存在一种解法只需要遍历一次，那不仅可以节省一些 CPU，还会使得代码更简洁，更具有可读性。

Scala 集合的 API 有一个叫做 `collect` 的方法，对于 `Seq[A]` ，它有如下方法签名：

```
def collect[B](pf: PartialFunction[A, B]): Seq[B]
```

这个方法将给定的 偏函数*(partial function)* 应用到序列的每一个元素上， 最后返回一个新的序列 - 偏函数做了 `filter` 和 `map` 要做的事情。

那偏函数到底是什么呢？ 概括来说，偏函数是一个一元函数，它只在部分输入上有定义， 并且允许使用者去检查其在一个给定的输入上是否有定义。 为此，特质 `PartialFunction` 提供了一个 `isDefinedAt` 方法。 事实上，类型 `PartialFunction[-A, +B]` 扩展了类型 `(A) => B` （一元函数，也可以写成 `Function1[A, B]` ）。 模式匹配型的匿名函数的类型就是 `PartialFunction` 。

依据继承关系，将一个模式匹配型的匿名函数传递给接受一元函数的方法（如： `map` 、 `filter` ）是没有问题的， 只要这个匿名函数对于所有可能的输入都有定义。

不过 `collect` 方法接受的函数只能是 `PartialFunction[A, B]` 类型的。 对于序列中的每一个元素，首先检查偏函数在其上面是否有定义， 如果没有定义，那这个元素就直接被忽略掉， 否则，就将偏函数应用到这个元素上，返回的结果加入结果集。

现在，我们来重构 `wordsWithoutOutliers` ，首先定义需要的偏函数：

```
val pf: PartialFunction[(String, Int), String] = {
  case (word, freq) if freq > 3 && freq < 25 => word
}
```

我们为这个案例加入了 守卫语句，不在区间里的元素就没有定义。

除了使用上面的这种方式，还可以显示的扩展 `PartialFunction` 特质：

```
val pf = new PartialFunction[(String, Int), String] {
  def apply(wordFrequency: (String, Int)) = wordFrequency match
{
    case (word, freq) if freq > 3 && freq < 25 => word
  }
  def isDefinedAt(wordFrequency: (String, Int)) = wordFrequency
match {
    case (word, freq) if freq > 3 && freq < 25 => true
    case _ => false
  }
}
```

当然，前一种方法更为更为简洁。

把定义好的 `pf` 传递给 `map` 函数，能够通过编译期，但运行时会抛出 `MatchError` ，因为我们的偏函数并不是在所有输入值上都有定义：

```
wordFrequencies.map(pf) // will throw a MatchError
```

不过，把它传递给 `collect` 函数就能得到想要的结果：

```
wordFrequencies.collect(pf) // List("habitual", "homely", "socie
ty")
```

这个结果和我们最初的实现所得到的结果是一样的，因此我们可以重写 `wordsWithoutOutliers` ：

```
def wordsWithoutOutliers(wordFrequencies: Seq[(String, Int)]): Seq[String] =
  wordFrequencies.collect { case (word, freq) if freq > 3 && freq < 25 => word }
```

偏函数还有其他一些有用的性质，比如说，它们可以被直接串联起来，实现函数式的 责任链模式 （源自于面向对象程式设计）。

偏函数还是很多 Scala 库和 API 的重要组成部分。比如： Akka 中，actor 处理信息的方法就是通过偏函数来定义的。 因此，理解这一概念是非常重要的。

## 小结

在这一章中，我们学习了另一种定义匿名函数的方法：一系列的匹配样例， 它用一种非常简洁的方式让解构数据成为可能。 而且，我们还深入到偏函数这个话题，用一个简单的例子展示了它的用处。

下一章，我们将深入的学习已经出现过的 `Option` 类型，探索其存在的原因及其使用方式。

# 类型 Option

前几章，我们讨论了许多相当先进的技术，尤其是模式匹配和提取器。 是时候来看一看 Scala 另一个基本特性了： Option 类型。

可能你已经见过它在 `Map` API 中的使用；在实现自己的提取器时，我们也用过它， 然而，它还需要更多的解释。 你可能会想知道它到底解决什么问题，为什么用它来处理缺失值要比其他方法好， 而且可能你还不知道该怎么在你的代码中使用它。 这一章的目的就是消除这些问号，并教授你作为一个新手所应该了解的 `Option` 知识。

## 基本概念

Java 开发者一般都知道 `NullPointerException` （其他语言也有类似的东西）， 通常这是由于某个方法返回了 `null` ，但这并不是开发者所希望发生的，代码也不好去处理这种异常。

值 `null` 通常被滥用来表征一个可能会缺失的值。 不过，某些语言以一种特殊的方法对待 `null` 值，或者允许你安全的使用可能是 `null` 的值。 比如说，Groovy 有 安全运算符*(Safe Navigation Operator)* 用于访问属性， 这样 `foo?.bar?.baz` 不会在 `foo` 或 `bar` 是 `null` 时而引发异常，而是直接返回 `null` ， 然而，Groovy 中没有什么机制来强制你使用此运算符，所以如果你忘记使用它，那就完蛋了！

Clojure 对待 `nil` 基本上就像对待空字符串一样。 也可以把它当作列表或者映射表一样去访问，这意味着， `nil` 在调用层级中向上冒泡。 很多时候这样是可行的，但有时会导致异常出现在更高的调用层级中，而那里的代码没有对 `nil` 加以考虑。

Scala 试图通过摆脱 `null` 来解决这个问题，并提供自己的类型用来表示一个值是可选的（有值或无值）， 这就是 `Option[A]` 特质。

`Option[A]` 是一个类型为 `A` 的可选值的容器： 如果值存在， `Option[A]` 就是一个 `Some[A]` ，如果不存在， `Option[A]` 就是对象 `None` 。

在类型层面上指出一个值是否存在，使用你的代码的开发者（也包括你自己）就会被编译器强制去处理这种可能性， 而不能依赖值存在的偶然性。

`Option` 是强制的！不要使用 `null` 来表示一个值是缺失的。

## 创建 Option

通常，你可以直接实例化 `Some` 样例类来创建一个 Option 。

```
val greeting: Option[String] = Some("Hello world")
```

或者，在知道值缺失的情况下，直接使用 `None` 对象：

```
val greeting: Option[String] = None
```

然而，在实际工作中，你不可避免的要去操作一些 Java 库， 或者是其他将 `null` 作为缺失值的JVM 语言的代码。 为此， `Option` 伴生对象提供了一个工厂方法，可以根据给定的参数创建相应的 `Option` ：

```
val absentGreeting: Option[String] = Option(null) // absentGreeting will be None
val presentGreeting: Option[String] = Option("Hello!") // presentGreeting will be Some("Hello!")
```

## 使用 Option

目前为止，所有的这些都很简洁，不过该怎么使用 Option 呢？是时候开始举些无聊的例子了。

想象一下，你正在为某个创业公司工作，要做的第一件事情就是实现一个用户的存储库， 要求能够通过唯一的用户 ID 来查找他们。 有时候请求会带来假的 ID，这种情况，查找方法就需要返回 `Option[User]` 类型的数据。 一个假想的实现可能是：

```
case class User(
  id: Int,
  firstName: String,
  lastName: String,
  age: Int,
  gender: Option[String]
)

object UserRepository {
  private val users = Map(1 -> User(1, "John", "Doe", 32, Some(
"male")),
                          2 -> User(2, "Johanna", "Doe", 30, N
one))
  def findById(id: Int): Option[User] = users.get(id)
  def findAll = users.values
}
```

现在，假设从 `UserRepository` 接收到一个 `Option[User]` 实例，并需要拿它做点什么，该怎么办呢？

一个办法就是通过 `isDefined` 方法来检查它是否有值。 如果有，你就可以用 `get` 方法来获取该值：

```
val user1 = UserRepository.findById(1)
if (user1.isDefined) {
  println(user1.get.firstName)
} // will print "John"
```

这和 Guava 库 中的 `Optional` 使用方法类似。 不过这种使用方式太过笨重，更重要的是，使用 `get` 之前， 你可能会忘记用 `isDefined` 做检查，这会导致运行期出现异常。 这样一来，相对于 `null` ，使用 `Option` 并没有什么优势。

你应该尽可能远离这种访问方式！

## 提供一个默认值

很多时候，在值不存在时，需要进行回退，或者提供一个默认值。 Scala 为 `Option` 提供了 `getOrElse` 方法，以应对这种情况：

```
val user = User(2, "Johanna", "Doe", 30, None)
println("Gender: " + user.gender.getOrElse("not specified")) // will print "not specified"
```

请注意，作为 `getOrElse` 参数的默认值是一个 传名参数 ， 这意味着，只有当这个 `Option` 确实是 `None` 时，传名参数才会被求值。 因此，没必要担心创建默认值的代价，它只有在需要时才会发生。

## 模式匹配

`Some` 是一个样例类，可以出现在模式匹配表达式或者其他允许模式出现的地方。 上面的例子可以用模式匹配来重写：

```
val user = User(2, "Johanna", "Doe", 30, None)
user.gender match {
  case Some(gender) => println("Gender: " + gender)
  case None => println("Gender: not specified")
}
```

或者，你想删除重复的 `println` 语句，并重点突出模式匹配表达式的使用：

```
val user = User(2, "Johanna", "Doe", 30, None)
val gender = user.gender match {
  case Some(gender) => gender
  case None => "not specified"
}
println("Gender: " + gender)
```

你可能已经发现用模式匹配处理 `Option` 实例是非常啰嗦的，这也是它非惯用法的原因。 所以，即使你很喜欢模式匹配，也尽量用其他方法吧。

不过在 Option 上使用模式确实是有一个相当优雅的方式， 在下面的 for 语句一节中，你就会学到。

# 作为集合的 Option

到目前为止，你还没有看见过优雅使用 Option 的方式吧。下面这个就是了。

前文我提到过， `Option` 是类型 `A` 的容器，更确切地说，你可以把它看作是某种集合， 这个特殊的集合要么只包含一个元素，要么就什么元素都没有。

虽然在类型层次上， `Option` 并不是 Scala 的集合类型， 但，凡是你觉得 Scala 集合好用的方法， `Option` 也有， 你甚至可以将其转换成一个集合，比如说 `List` 。

那么这又能让你做什么呢？

## 执行一个副作用

如果想在 Option 值存在的时候执行某个副作用， `foreach` 方法就派上用场了：

```
UserRepository.findById(2).foreach(user => println(user.firstName)) // prints "Johanna"
```

如果这个 Option 是一个 `Some` ，传递给 `foreach` 的函数就会被调用一次，且只有一次； 如果是 `None` ，那它就不会被调用。

## 执行映射

`Option` 表现的像集合，最棒的一点是， 你可以用它来进行函数式编程，就像处理列表、集合那样。

正如你可以将 `List[A]` 映射到 `List[B]` 一样，你也可以映射 `Option[A]` 到 `Option[B]` ： 如果 `Option[A]` 实例是 `Some[A]` 类型，那映射结果就是 `Some[B]` 类型；否则，就是 `None` 。

如果将 `Option` 和 `List` 做对比 ，那 `None` 就相当于一个空列表： 当你映射一个空的 `List[A]` ，会得到一个空的 `List[B]` ， 而映射一个是 `None` 的 `Option[A]` 时，得到的 `Option[B]` 也是 `None` 。

让我们得到一个可能不存在的用户的年龄：

```
val age = UserRepository.findById(1).map(_.age) // age is Some(3
2)
```

## Option 与 flatMap

也可以在 `gender` 上做 `map` 操作：

```
val gender = UserRepository.findById(1).map(_.gender) // gender
is an Option[Option[String]]
```

所生成的 `gender` 类型是 `Option[Option[String]]` 。这是为什么呢？

这样想：你有一个装有 `User` 的 `Option` 容器，在容器里面，你将 `User` 映射到 `Option[String]` （ `User` 类上的属性 `gender` 是 `Option[String]` 类型的）。 得到的必然是嵌套的 Option。

既然可以 `flatMap` 一个 `List[List[A]]` 到 `List[B]` ，也可以 `flatMap` 一个 `Option[Option[A]]` 到 `Option[B]` ，这没有任何问题： Option 提供了 `flatMap` 方法。

```
val gender1 = UserRepository.findById(1).flatMap(_.gender) // ge
nder is Some("male")
val gender2 = UserRepository.findById(2).flatMap(_.gender) // ge
nder is None
val gender3 = UserRepository.findById(3).flatMap(_.gender) // ge
nder is None
```

现在结果就变成了 `Option[String]` 类型， 如果 `user` 和 `gender` 都有值，那结果就会是 `Some` 类型，反之，就得到一个 `None` 。

要理解这是什么原理，让我们看看当 `flatMap` 一个 `List[List[A]]` 时，会发生什么？ （要记得， Option 就像一个集合，比如列表）

```
val names: List[List[String]] =
 List(List("John", "Johanna", "Daniel"), List(), List("Doe", "We
stheide"))
names.map(_.map(_.toUpperCase))
// results in List(List("JOHN", "JOHANNA", "DANIEL"), List(), Li
st("DOE", "WESTHEIDE"))
names.flatMap(_.map(_.toUpperCase))
// results in List("JOHN", "JOHANNA", "DANIEL", "DOE", "WESTHEID
E")
```

如果我们使用 `flatMap` ，内部列表中的所有元素会被转换成一个扁平的字符串列表。显然，如果内部列表是空的，则不会有任何东西留下。

现在回到 `Option` 类型，如果映射一个由 `Option` 组成的列表呢？

```
val names: List[Option[String]] = List(Some("Johanna"), None, So
me("Daniel"))
names.map(_.map(_.toUpperCase)) // List(Some("JOHANNA"), None, S
ome("DANIEL"))
names.flatMap(xs => xs.map(_.toUpperCase)) // List("JOHANNA", "D
ANIEL")
```

如果只是 `map` ，那结果类型还是 `List[Option[String]]` 。而使用 `flatMap` 时，内部集合的元素就会被放到一个扁平的列表里：任何一个 `Some[String]` 里的元素都会被解包，放入结果集中；而原列表中的 `None` 值由于不包含任何元素，就直接被过滤出去了。

记住这一点，然后再去看看 `faltMap` 在 `Option` 身上做了什么。

## 过滤 Option

也可以像过滤列表那样过滤 Option：如果选项包含有值，而且传递给 `filter` 的谓词函数返回真， `filter` 会返回 `Some` 实例。否则（即选项没有值，或者谓词函数返回假值），返回值为 `None` 。

```
UserRepository.findById(1).filter(_.age > 30) // None, because age is <= 30
UserRepository.findById(2).filter(_.age > 30) // Some(user), because age is > 30
UserRepository.findById(3).filter(_.age > 30) // None, because user is already None
```

## for 语句

现在，你已经知道 Option 可以被当作集合来看待，并且有 `map` 、 `flatMap` 、 `filter` 这样的方法。 可能你也在想 Option 是否能够用在 for 语句中，答案是肯定的。 而且，用 for 语句来处理 Option 是可读性最好的方式，尤其是当你有多个 `map` 、 `flatMap` 、 `filter` 调用的时候。 如果只是一个简单的 `map` 调用，那 for 语句可能有点繁琐。

假如我们想得到一个用户的性别，可以这样使用 for 语句：

```
for {
  user <- UserRepository.findById(1)
  gender <- user.gender
} yield gender // results in Some("male")
```

可能你已经知道，这样的 for 语句等同于嵌套的 `flatMap` 调用。 如果 `UserRepository.findById` 返回 `None` ，或者 `gender` 是 `None` ，那这个 for 语句的结果就是 `None` 。 不过这个例子里， `gender` 含有值，所以返回结果是 `Some` 类型的。

如果我们想返回所有用户的性别（当然，如果用户设置了性别），可以遍历用户，yield 其性别：

```
for {
  user <- UserRepository.findAll
  gender <- user.gender
} yield gender
// result in List("male")
```

### 在生成器左侧使用

也许你还记得，前一章曾经提到过， for 语句中生成器的左侧也是一个模式。 这意味着也可以在 for 语句中使用包含选项的模式。

重写之前的例子：

```
for {
  User(_, _, _, _, Some(gender)) <- UserRepository.findAll
} yield gender
```

在生成器左侧使用 `Some` 模式就可以在结果集中排除掉值为 `None` 的元素。

## 链接 Option

Option 还可以被链接使用，这有点像偏函数的链接： 在 Option 实例上调用 `orElse` 方法，并将另一个 Option 实例作为传名参数传递给它。 如果一个 Option 是 `None` ， `orElse` 方法会返回传名参数的值，否则，就直接返回这个 Option。

一个很好的使用案例是资源查找：对多个不同的地方按优先级进行搜索。 下面的例子中，我们首先搜索 *config* 文件夹，并调用 `orElse` 方法，以传递备用目录：

```
case class Resource(content: String)
val resourceFromConfigDir: Option[Resource] = None
val resourceFromClasspath: Option[Resource] = Some(Resource("I w
as found on the classpath"))
val resource = resourceFromConfigDir orElse resourceFromClasspat
h
```

如果想链接多个选项，而不仅仅是两个，使用 `orElse` 会非常合适。 不过，如果只是想在值缺失的情况下提供一个默认值，那还是使用 `getOrElse` 吧。

### 总结

在这一章里，你学到了有关 Option 的所有知识， 这有利于你理解别人的代码，也有利于你写出更可读，更函数式的代码。

这一章最重要的一点是：列表、集合、映射、Option，以及之后你会见到的其他数据类型， 它们都有一个非常统一的使用方式，这种使用方式既强大又优雅。

下一章，你将学习 Scala 错误处理的惯用法。

# Try 与错误处理

当你在尝试一门新的语言时，可能不会过于关注程序出错的问题， 但当真的去创造可用的代码时，就不能再忽视代码中的可能产生的错误和异常了。 鉴于各种各样的原因，人们往往低估了语言对错误处理支持程度的重要性。

事实会表明，Scala 能够很优雅的处理此类问题， 这一部分，我会介绍 Scala 基于 Try 的错误处理机制，以及这背后的原因。 我将使用一个在 *Scala 2.10* 新引入的特性，该特性向 *2.9.3* 兼容， 因此，请确保你的 Scala 版本不低于 *2.9.3*。

## 异常的抛出和捕获

在介绍 Scala 错误处理的惯用法之前，我们先看看其他语言（如，Java，Ruby）的错误处理机制。 和这些语言类似，Scala 也允许你抛出异常：

```
case class Customer(age: Int)
class Cigarettes
case class UnderAgeException(message: String) extends Exception(
message)
def buyCigarettes(customer: Customer): Cigarettes =
  if (customer.age < 16)
    throw UnderAgeException(s"Customer must be older than 16 but
 was ${customer.age}")
  else new Cigarettes
```

被抛出的异常能够以类似 Java 中的方式被捕获，虽然是使用偏函数来指定要处理的异常类型。 此外，Scala 的 `try/catch` 是表达式（返回一个值），因此下面的代码会返回异常的消息：

```
val youngCustomer = Customer(15)
try {
  buyCigarettes(youngCustomer)
  "Yo, here are your cancer sticks! Happy smokin'!"
} catch {
    case UnderAgeException(msg) => msg
}
```

## 函数式的错误处理

现在，如果代码中到处是上面的异常处理代码，那它很快就会变得丑陋无比，和函数式程序设计非常不搭。 对于高并发应用来说，这也是一个很差劲的解决方式，比如， 假设需要处理在其他线程执行的 actor 所引发的异常，显然你不能用捕获异常这种处理方式， 你可能会想到其他解决方案，例如去接收一个表示错误情况的消息。

一般来说，在 Scala 中，好的做法是通过从函数里返回一个合适的值来通知人们程序出错了。 别担心，我们不会回到 C 中那种需要使用按约定进行检查的错误编码的错误处理。 相反，Scala 使用一个特定的类型来表示可能会导致异常的计算，这个类型就是 Try。

### Try 的语义

解释 Try 最好的方式是将它与上一章所讲的 Option 作对比。

`Option[A]` 是一个可能有值也可能没值的容器， `Try[A]` 则表示一种计算： 这种计算在成功的情况下，返回类型为 `A` 的值，在出错的情况下，返回 `Throwable` 。 这种可以容纳错误的容器可以很轻易的在并发执行的程序之间传递。

Try 有两个子类型：

1. `Success[A]` ：代表成功的计算。
2. 封装了 `Throwable` 的 `Failure[A]` ：代表出了错的计算。

如果知道一个计算可能导致错误，我们可以简单的使用 `Try[A]` 作为函数的返回类型。 这使得出错的可能性变得很明确，而且强制客户端以某种方式处理出错的可能。

假设，需要实现一个简单的网页爬取器：用户能够输入想爬取的网页 URL， 程序就需要去分析 URL 输入，并从中创建一个 `java.net.URL` ：

```
import scala.util.Try
import java.net.URL
def parseURL(url: String): Try[URL] = Try(new URL(url))
```

正如你所看到的，函数返回类型为 `Try[URL]` ：如果给定的 url 语法正确，这将是 `Success[URL]` ，否则， `URL` 构造器会引发 `MalformedURLException` ，从而返回值变成 `Failure[URL]` 类型。

上例中，我们还用了 Try 伴生对象里的 `apply` 工厂方法，这个方法接受一个类型为 `A` 的传名参数，这意味着， `new URL(url)` 是在 `Try` 的 `apply` 方法里执行的。

`apply` 方法会捕获任何非致命的异常，返回一个包含相关异常的 Failure 实例。

因此， `parseURL("http://danielwestheide.com")` 会返回一个 `Success[URL]` ，包含了解析后的网址，而 `parseULR("garbage")` 将返回一个含有 `MalformedURLException` 的 `Failure[URL]` 。

## 使用 Try

使用 Try 与使用 Option 非常相似，在这里你看不到太多新的东西。

你可以调用 `isSuccess` 方法来检查一个 Try 是否成功，然后通过 `get` 方法获取它的值，但是，这种方式的使用并不多见，因为你可以用 `getOrElse` 方法给 Try 提供一个默认值：

```
val url = parseURL(Console.readLine("URL: ")) getOrElse new URL(
"http://duckduckgo.com")
```

如果用户提供的 URL 格式不正确，我们就使用 DuckDuckGo 的 URL 作为备用。

## 链式操作

Try 最重要的特征是，它也支持高阶函数，就像 Option 一样。 在下面的示例中，你将看到，在 Try 上也进行链式操作，捕获可能发生的异常，而且代码可读性不错。

## Mapping 和 Flat Mapping

将一个是 `Success[A]` 的 `Try[A]` 映射到 `Try[B]` 会得到 `Success[B]` 。如果它是 `Failure[A]` ，就会得到 `Failure[B]` ，而且包含的异常和 `Failure[A]` 一样。

```
parseURL("http://danielwestheide.com").map(_.getProtocol)
// results in Success("http")
parseURL("garbage").map(_.getProtocol)
// results in Failure(java.net.MalformedURLException: no protoco
l: garbage)
```

如果链接多个 `map` 操作，会产生嵌套的 Try 结构，这并不是我们想要的。 考虑下面这个返回输入流的方法：

```
import java.io.InputStream
def inputStreamForURL(url: String): Try[Try[Try[InputStream]]] =
 parseURL(url).map { u =>
 Try(u.openConnection()).map(conn => Try(conn.getInputStream))
}
```

由于每个传递给 `map` 的匿名函数都返回 Try，因此返回类型就变成了 `Try[Try[Try[InputStream]]]` 。

这时候， `flatMap` 就派上用场了。 `Try[A]` 上的 `flatMap` 方法接受一个映射函数，这个函数类型是 `(A) => Try[B]` 。 如果我们的 `Try[A]` 已经是 `Failure[A]` 了，那么里面的异常就直接被封装成 `Failure[B]` 返回， 否则， `flatMap` 将 `Success[A]` 里面的值解包出来，并通过映射函数将其映射到 `Try[B]` 。

这意味着，我们可以通过链接任意个 `flatMap` 调用来创建一条操作管道，将值封装在 Success 里一层层的传递。

现在让我们用 `flatMap` 来重写先前的例子：

```
def inputStreamForURL(url: String): Try[InputStream] =
 parseURL(url).flatMap { u =>
   Try(u.openConnection()).flatMap(conn => Try(conn.getInputStre
am))
 }
```

这样，我们就得到了一个 `Try[InputStream]` ，它可以是一个 Failure，包含了在 `flatMap` 过程中可能出现的异常；也可以是一个 Success，包含了最后的结果。

### 过滤器和 foreach

当然，你也可以对 Try 进行过滤，或者调用 `foreach` ，既然已经学过 Option，对于这两个方法也不会陌生。

当一个 Try 已经是 `Failure` 了，或者传递给它的谓词函数返回假值， `filter` 就返回 `Failure` （如果是谓词函数返回假值，那 `Failure` 里包含的异常是 `NoSuchException` ），否则的话， `filter` 就返回原本的那个 `Success` ，什么都不会变：

```
def parseHttpURL(url: String) = parseURL(url).filter(_.getProtocol == "http")
parseHttpURL("http://apache.openmirror.de") // results in a Success[URL]
parseHttpURL("ftp://mirror.netcologne.de/apache.org") // results in a Failure[URL]
```

当一个 Try 是 `Success` 时， `foreach` 允许你在被包含的元素上执行副作用，这种情况下，传递给 `foreach` 的函数只会执行一次，毕竟 Try 里面只有一个元素：

```
parseHttpURL("http://danielwestheide.com").foreach(println)
```

> 当 Try 是 Failure 时， `foreach` 不会执行，返回 `Unit` 类型。

## for 语句中的 Try

既然 Try 支持 `flatMap` 、 `map` 、 `filter` ，能够使用 for 语句也是理所当然的事情，而且这种情况下的代码更可读。为了证明这一点，我们来实现一个返回给定 URL 的网页内容的函数：

```
import scala.io.Source
def getURLContent(url: String): Try[Iterator[String]] =
  for {
   url <- parseURL(url)
   connection <- Try(url.openConnection())
   is <- Try(connection.getInputStream)
   source = Source.fromInputStream(is)
  } yield source.getLines()
```

这个方法中，有三个可能会出错的地方，但都被 Try 给涵盖了。 第一个是我们已经实现的 `parseURL` 方法， 只有当它是一个 `Success[URL]` 时，我们才会尝试打开连接，从中创建一个新的 `InputStream` 。 如果这两步都成功了，我们就 `yield` 出网页内容，得到的结果是 `Try[Iterator[String]]` 。

当然，你可以使用 `Source#fromURL` 简化这个代码，并且，这个代码最后没有关闭输入流， 这都是为了保持例子的简单性，专注于要讲述的主题。

> 在这个例子中， `Source#fromURL` 可以这样用：
>
> ```
> import scala.io.Source
> def getURLContent(url: String): Try[Iterator[String]] =
>   for {
>     url <- parseURL(url)
>     source = Source.fromURL(url)
>   } yield source.getLines()
> ```
>
> 用 `is.close()` 可以关闭输入流。

## 模式匹配

代码往往需要知道一个 Try 实例是 Success 还是 Failure，这时候，你应该想到模式匹配， 也幸好， `Success` 和 `Failure` 都是样例类。

接着上面的例子，如果网页内容能顺利提取到，我们就展示它，否则，打印一个错误信息：

```
import scala.util.Success
import scala.util.Failure
getURLContent("http://danielwestheide.com/foobar") match {
  case Success(lines) => lines.foreach(println)
  case Failure(ex) => println(s"Problem rendering URL content: $
{ex.getMessage}")
}
```

## 从故障中恢复

如果想在失败的情况下执行某种动作，没必要去使用 `getOrElse` ，一个更好的选择是 `recover` ，它接受一个偏函数，并返回另一个 Try。 如果 `recover` 是在 Success 实例上调用的，那么就直接返回这个实例，否则就调用偏函数。 如果偏函数为给定的 `Failure` 定义了处理动作， `recover` 会返回 `Success` ，里面包含偏函数运行得出的结果。

下面是应用了 `recover` 的代码：

```
import java.net.MalformedURLException
import java.io.FileNotFoundException
val content = getURLContent("garbage") recover {
  case e: FileNotFoundException => Iterator("Requested page does
 not exist")
  case e: MalformedURLException => Iterator("Please make sure to
 enter a valid URL")
  case _ => Iterator("An unexpected error has occurred. We are s
o sorry!")
}
```

现在，我们可以在返回值 `content` 上安全的使用 `get` 方法了，因为它一定是一个 Success。 调用 `content.get.foreach(println)` 会打印 *Please make sure to enter a valid URL*。

## 总结

Scala 的错误处理和其他范式的编程语言有很大的不同。 Try 类型可以让你将可能会出错的计算封装在一个容器里，并优雅的去处理计算得到的值。 并且可以像操作集合和 Option 那样统一的去操作 Try。

Try 还有其他很多重要的方法，鉴于篇幅限制，这一章并没有全部列出，比如 `orElse` 方法， `transform` 和 `recoverWith` 也都值得去看。

下一章，我们会探讨 Either，另外一种可以代表计算的类型，但它的可使用范围要比 Try 大的多。

# 类型 Either

上一章介绍了 Try，它用函数式风格来处理程序错误。 这一章我们介绍一个和 Try 相似的类型 - Either， 学习如何去使用它，什么时候去使用它，以及它有什么缺点。

不过首先得知道一件事情： 在写作这篇文章的时候，Either 有一些设计缺陷，很多人都在争论到底要不要使用它。 既然如此，为什么还要学习它呢？ 因为，在理解 Try 这个错综复杂的类型之前，不是所有人都会在代码中使用 Try 风格的异常处理。 其次，Try 不能完全替代 Either，它只是 Either 用来处理异常的一个特殊用法。 Try 和 Either 互相补充，各自侧重于不同的使用场景。

因此，尽管 Either 有缺陷，在某些情况下，它依旧是非常合适的选择。

## Either 语义

Either 也是一个容器类型，但不同于 Try、Option，它需要两个类型参数： `Either[A, B]` 要么包含一个类型为 `A` 的实例，要么包含一个类型为 `B` 的实例。 这和 `Tuple2[A, B]` 不一样， `Tuple2[A, B]` 是两者都要包含。

Either 只有两个子类型： Left、 Right， 如果 `Either[A, B]` 对象包含的是 `A` 的实例，那它就是 Left 实例，否则就是 Right 实例。

在语义上，Either 并没有指定哪个子类型代表错误，哪个代表成功， 毕竟，它是一种通用的类型，适用于可能会出现两种结果的场景。 而异常处理只不过是其一种常见的使用场景而已， 不过，按照约定，处理异常时，Left 代表出错的情况，Right 代表成功的情况。

## 创建 Either

创建 Either 实例非常容易，Left 和 Right 都是样例类。 要是想实现一个“坚如磐石”的互联网审查程序，可以直接这么做：

```
import scala.io.Source
import java.net.URL
def getContent(url: URL): Either[String, Source] =
  if(url.getHost.contains("google"))
    Left("Requested URL is blocked for the good of the people!")
  else
    Right(Source.fromURL(url))
```

调用 `getContent(new URL("http://danielwestheide.com"))` 会得到一个封装有 `scala.io.Source` 实例的 Right，传入 `new URL("https://plus.google.com")` 会得到一个含有 `String` 的 Left。

## Either 用法

Either 基本的使用方法和 Option、Try 一样：调用 `isLeft` （或 `isRight` ）方法询问一个 Either，判断它是 Left 值，还是 Right 值。可以使用模式匹配，这是最方便也是最为熟悉的一种方法：

```
getContent(new URL("http://google.com")) match {
  case Left(msg) => println(msg)
  case Right(source) => source.getLines.foreach(println)
}
```

### 立场

你不能，至少不能直接像 Option、Try 那样把 Either 当作一个集合来使用，因为 Either 是 无偏**(unbiased)** 的。

Try 偏向 Success： `map` 、 `flatMap` 以及其他一些方法都假设 Try 对象是一个 Success 实例，如果是 Failure，那这些方法不做任何事情，直接将这个 Failure 返回。

但 Either 不做任何假设，这意味着首先你要选择一个立场，假设它是 Left 还是 Right，然后在这个假设的前提下拿它去做你想做的事情。调用 `left` 或 `right` 方法，就能得到 Either 的 `LeftProjection` 或 `RightProjection` 实例，这就是 Either 的 立场*(Projection)* ，它们是对 Either 的一个左偏向的或右偏向的封装。

## 映射

一旦有了 Projection，就可以调用 `map` ：

```
val content: Either[String, Iterator[String]] =
  getContent(new URL("http://danielwestheide.com")).right.map(_.
getLines())
// content is a Right containing the lines from the Source retur
ned by getContent
val moreContent: Either[String, Iterator[String]] =
  getContent(new URL("http://google.com")).right.map(_.getLines)
// moreContent is a Left, as already returned by getContent

// content: Either[String,Iterator[String]] = Right(non-empty it
erator)
// moreContent: Either[String,Iterator[String]] = Left(Requested
 URL is blocked for the good of the people!)
```

这个例子中，无论 `Either[String, Source]` 是 Left 还是 Right， 它都会被映射到 `Either[String, Iterator[String]]` 。 如果，它是一个 Right 值，这个值就会被 `_.getLines()` 转换； 如果，它是一个 Left 值，就直接返回这个值，什么都不会改变。

LeftProjection也是类似的：

```
val content: Either[Iterator[String], Source] =
  getContent(new URL("http://danielwestheide.com")).left.map(Ite
rator(_))
// content is the Right containing a Source, as already returned
 by getContent
val moreContent: Either[Iterator[String], Source] =
  getContent(new URL("http://google.com")).left.map(Iterator(_))
// moreContent is a Left containing the msg returned by getConte
nt in an Iterator

// content: Either[Iterator[String],scala.io.Source] = Right(non
-empty iterator)
// moreContent: Either[Iterator[String],scala.io.Source] = Left(
non-empty iterator)
```

现在，如果 Either 是个 Left 值，里面的值会被转换；如果是 Right 值，就维持原样。两种情况下，返回类型都是 `Either[Iterator[String, Source]` 。

> 请注意， `map` 方法是定义在 Projection 上的，而不是 Either，但其返回类型是 Either，而不是 Projection。
>
> 可以看到，Either 和其他你知道的容器类型之所以不一样，就是因为它的无偏性。接下来你会发现，在特定情况下，这会产生更多的麻烦。而且，如果你想在一个 Either 上多次调用 `map` 、 `flatMap` 这样的方法，你总需要做 Projection，去选择一个立场。

## Flat Mapping

Projection 也支持 flat mapping，避免了嵌套使用 map 所造成的令人费解的类型结构。

假设我们想计算两篇文章的平均行数，下面的代码可以解决这个“富有挑战性”的问题：

```
val part5 = new URL("http://t.co/UR1aalX4")
val part6 = new URL("http://t.co/6wlKwTmu")
val content = getContent(part5).right.map(a =>
  getContent(part6).right.map(b =>
    (a.getLines().size + b.getLines().size) / 2))
// => content: Product with Serializable with scala.util.Either[
String,Product with Serializable with scala.util.Either[String,I
nt]] = Right(Right(537))
```

运行上面的代码，会得到什么？会得到一个类型为 `Either[String, Either[String, Int]]` 的玩意儿。当然，你可以调用 `joinRight` 方法来使得这个结果 扁平化**(flatten)** 。

不过我们可以直接避免这种嵌套结构的产生，如果在最外层的 RightProjection 上调用 `flatMap` 函数，而不是 `map` ，得到的结果会更好看些，因为里层 Either 的值被解包了：

```
val content = getContent(part5).right.flatMap(a =>
  getContent(part6).right.map(b =>
    (a.getLines().size + b.getLines().size) / 2))
// => content: scala.util.Either[String,Int] = Right(537)
```

现在， `content` 值类型变成了 `Either[String, Int]` ，处理它相对来说就很容易了。

## for 语句

说到 for 语句，想必现在，你应该已经爱上它在不同类型上的一致性表现了。 在 for 语句中，也能够使用 `Either` 的 Projection，但遗憾的是，这样做需要一些丑陋的变通。

假设用 for 语句重写上面的例子：

```
def averageLineCount(url1: URL, url2: URL): Either[String, Int]
=
  for {
    source1 <- getContent(url1).right
    source2 <- getContent(url2).right
  } yield (source1.getLines().size + source2.getLines().size) / 2
```

这个代码还不是太坏，毕竟只需要额外调用 `left` 、 `right` 。

但是你不觉得 yield 语句太长了吗？现在，我就把它移到值定义块中：

```
def averageLineCountWontCompile(url1: URL, url2: URL): Either[St
ring, Int] =
  for {
    source1 <- getContent(url1).right
    source2 <- getContent(url2).right
    lines1 = source1.getLines().size
    lines2 = source2.getLines().size
  } yield (lines1 + lines2) / 2
```

试着去编译它，然后你会发现无法编译！如果我们把 for 语法糖去掉，原因可能会清晰些。展开上面的代码得到：

```
def averageLineCountDesugaredWontCompile(url1: URL, url2: URL):
Either[String, Int] =
  getContent(url1).right.flatMap { source1 =>
    getContent(url2).right.map { source2 =>
      val lines1 = source1.getLines().size
      val lines2 = source2.getLines().size
      (lines1, lines2)
    }.map { case (x, y) => x + y / 2 }
  }
```

问题在于，在 for 语句中追加新的值定义会在前一个 `map` 调用上自动引入另一个 `map` 调用，前一个 `map` 调用返回的是 Either 类型，不是 RightProjection 类型，而 Scala 并没有在 Either 上定义 `map` 函数，因此编译时会出错。

这就是 Either 丑陋的一面。要解决这个例子中的问题，可以不添加新的值定义。但有些情况，就必须得添加，这时候可以将值封装成 Either 来解决这个问题：

```
def averageLineCount(url1: URL, url2: URL): Either[String, Int]
=
  for {
    source1 <- getContent(url1).right
    source2 <- getContent(url2).right
    lines1 <- Right(source1.getLines().size).right
    lines2 <- Right(source2.getLines().size).right
  } yield (lines1 + lines2) / 2
```

认识到这些设计缺陷是非常重要的，这不会影响 Either 的可用性，但如果不知道发生了什么，它会让你感到非常头痛。

## 其他方法

Projection 还有其他有用的方法：

1. 可以在 Either 的某个 Projection 上调用 `toOption` 方法，将其转换成 Option。

假如，你有一个类型为 `Either[A, B]` 的实例 `e` ，`e.right.toOption` 会返回一个 `Option[B]` 。如果 `e` 是一个 Right 值，那这个 `Option[B]` 会是 Some 类型，如果 `e` 是一个 Left 值，那 `Option[B]` 就会是 `None` 。调用 `e.left.toOption` 也会有相应的结果。

2. 还可以用 `toSeq` 方法将 Either 转换为序列。

## Fold 函数

如果想变换一个 Either（不论它是 Left 值还是 right 值），可以使用定义在 Either 上的 `fold` 方法。这个方法接受两个返回相同类型的变换函数，当这个 Either 是 Left 值时，第一个函数会被调用；否则，第二个函数会被调用。

为了说明这一点，我们用 `fold` 重写之前的一个例子：

```
val content: Iterator[String] =
  getContent(new URL("http://danielwestheide.com")).fold(Iterator(_), _.getLines())
val moreContent: Iterator[String] =
  getContent(new URL("http://google.com")).fold(Iterator(_), _.getLines())
```

这个示例中，我们把 `Either[String, String]` 变换成了 `Iterator[String]` 。当然，你也可以在变换函数里返回一个新的 Either，或者是只执行副作用。`fold` 是一个可以用来替代模式匹配的好方法。

# 何时使用 Either

知道了 Either 的用法和应该注意的事项，我们来看看一些特殊的用例。

## 错误处理

可以用 Either 来处理异常，就像 Try 一样。不过 Either 有一个优势：可以使用更为具体的错误类型，而 Try 只能用 `Throwable` 。（这表明 Either 在处理自定义的错误时是个不错的选择）不过，需要实现一个方法，将这个功能委托给 `scala.util.control` 包中的 `Exception` 对象：

```
import scala.util.control.Exception.catching
def handling[Ex <: Throwable, T](exType: Class[Ex])(block: => T)
: Either[Ex, T] =
  catching(exType).either(block).asInstanceOf[Either[Ex, T]]
```

这么做的原因是，虽然 `scala.util.Exception` 提供的方法允许你捕获某些类型的异常，但编译期产生的类型总是 `Throwable` ，因此需要使用 `asInstanceOf` 方法强制转换。

有了这个方法，就可以把期望要处理的异常类型，放在 Either 里了：

```
import java.net.MalformedURLException
def parseURL(url: String): Either[MalformedURLException, URL] =
  handling(classOf[MalformedURLException])(new URL(url))
```

`handling` 的第二个参数 `block` 中可能还会有其他产生错误的情形，而且并不是所有情形都会抛出异常。这种情况下，没必要为了捕获异常而人为抛出异常，相反，只需定义你自己的错误类型，最好是样例类，并在错误情况发生时返回一个封装了这个类型实例的 Left。

下面是一个例子：

```
case class Customer(age: Int)
class Cigarettes
case class UnderAgeFailure(age: Int, required: Int)
def buyCigarettes(customer: Customer): Either[UnderAgeFailure, C
igarettes] =
  if (customer.age < 16) Left(UnderAgeFailure(customer.age, 16))
  else Right(new Cigarettes)
```

应该避免使用 Either 来封装意料之外的异常，使用 Try 来做这种事情会更好，至少它没有 Either 这样那样的缺陷。

## 处理集合

有些时候，当按顺序依次处理一个集合时，里面的某个元素产生了意料之外的结果， 但是这时程序不应该直接引发异常，因为这样会使得剩下的元素无法处理。Either 也非常适用于这种情况。

假设，在我们“行业标准般的” Web 审查系统里，使用了某种黑名单：

```
type Citizen = String
case class BlackListedResource(url: URL, visitors: Set[Citizen])

val blacklist = List(
  BlackListedResource(new URL("https://google.com"), Set("John D
oe", "Johanna Doe")),
  BlackListedResource(new URL("http://yahoo.com"), Set.empty),
  BlackListedResource(new URL("https://maps.google.com"), Set("J
ohn Doe")),
  BlackListedResource(new URL("http://plus.google.com"), Set.emp
ty)
)
```

`BlackListedResource` 表示黑名单里的网站 URL，外加试图访问这个网址的公民集合。

现在我们想处理这个黑名单，为了标识“有问题”的公民，比如说那些试图访问被屏蔽网站的人。 同时，我们想确定可疑的 Web 网站：如果没有一个公民试图去访问黑名单里的某一个网站， 那么就必须假定目标对象因为一些我们不知道的原因绕过了筛选器，需要对此进行调查。

下面的代码展示了该如何处理黑名单的：

```
val checkedBlacklist: List[Either[URL, Set[Citizen]]] =
  blacklist.map(resource =>
    if (resource.visitors.isEmpty) Left(resource.url)
    else Right(resource.visitors))
```

我们创建了一个 Either 序列，其中 `Left` 实例代表可疑的 URL， `Right` 是问题市民的集合。 识别问题公民和可疑网站变得非常简单。

```
val suspiciousResources = checkedBlacklist.flatMap(_.left.toOpti
on)
val problemCitizens = checkedBlacklist.flatMap(_.right.toOption)
.flatten.toSet
```

`Either` 非常适用于这种比异常处理更为普通的使用场景。

## 总结

目前为止，你应该已经学会了怎么使用 Either，认识到它的缺陷，以及知道该在什么时候用它。 鉴于 Either 的缺陷，使用不使用它，全都取决于你。 其实在实践中，你会注意到，有了 Try 之后，Either 不会出现那么多糟糕的使用情形。

不管怎样，分清楚它带来的利与弊总没有坏处。

# 类型 Future

作为一个对 Scala 充满热情的开发者，你应该已经听说过 Scala 处理并发的能力，或许你就是被这个吸引来的。 相较于大多数编程语言低级的并发 API，Scala 提供的方法可以让人们更好的理解并发以及编写良构的并发程序。

本章的主题- Future 就是这种方法的两大基石之一。（另一个是 actor） 我会解释 Future 的优点，以及它的函数式特征。

如果你想动手试试接下来的例子，请确保 Scala 版本不低于 2.9.3， Future 在 2.10.0 版本中引入，并向后兼容到 2.9.3，最初，它是 Akka 库的一部分（API略有不同）。

## 顺序代码为什么会变坏

假设你想准备一杯卡布奇诺，你可以一个接一个的执行以下步骤：

1. 研磨所需的咖啡豆
2. 加热一些水
3. 用研磨好的咖啡豆和热水制做一杯咖啡
4. 打奶泡
5. 结合咖啡和奶泡做成卡布奇诺

转换成 Scala 代码，可能会是这样：

```
import scala.util.Try
// Some type aliases, just for getting more meaningful method si
gnatures:
type CoffeeBeans = String
type GroundCoffee = String
case class Water(temperature: Int)
type Milk = String
type FrothedMilk = String
type Espresso = String
type Cappuccino = String

// dummy implementations of the individual steps:
def grind(beans: CoffeeBeans): GroundCoffee = s"ground coffee of
```

```
$beans"
def heatWater(water: Water): Water ` water.copy(temperature `  85)
def frothMilk(milk: Milk): FrothedMilk = s"frothed $milk"
def brew(coffee: GroundCoffee, heatedWater: Water): Espresso = "espresso"
def combine(espresso: Espresso, frothedMilk: FrothedMilk): Cappuccino = "cappuccino"

// some exceptions for things that might go wrong in the individual steps
// (we'll need some of them later, use the others when experimenting with the code):
case class GrindingException(msg: String) extends Exception(msg)
case class FrothingException(msg: String) extends Exception(msg)
case class WaterBoilingException(msg: String) extends Exception(msg)
case class BrewingException(msg: String) extends Exception(msg)

// going through these steps sequentially:
def prepareCappuccino(): Try[Cappuccino] = for {
  ground <- Try(grind("arabica beans"))
  water <- Try(heatWater(Water(25)))
  espresso <- Try(brew(ground, water))
  foam <- Try(frothMilk("milk"))
} yield combine(espresso, foam)
```

这样做有几个优点： 可以很轻易的弄清楚事情的步骤，一目了然，而且不会混淆。（毕竟没有上下文切换） 不好的一面是，大部分时间，你的大脑和身体都处于等待的状态： 在等待研磨咖啡豆时，你完全不能做任何事情，只有当这一步完成后，你才能开始烧水。 这显然是在浪费时间，所以你可能想一次开始多个步骤，让它们同时执行， 一旦水烧开，咖啡豆也磨好了，你可以制做咖啡了，这期间，打奶泡也可以开始了。

这和编写软件没什么不同。 一个 Web 服务器可以用来处理和响应请求的线程只有那么多， 不能因为要等待数据库查询或其他 HTTP 服务调用的结果而阻塞了这些可贵的线程。 相反，一个异步编程模型和非阻塞 IO 会更合适， 这样的话，当一个请求处理在等待数据库查询结果时，处理这个请求的线程也能够为其他请求服务。

> "I heard you like callbacks, so I put a callback in your callback!"

在并发家族里，你应该已经知道 nodejs 这个很酷的家伙，nodejs 完全通过回调来通信， 不幸的是，这很容易导致回调中包含回调的回调，这简直是一团糟，代码难以阅读和调试。

Scala 的 Future 也允许回调，但它提供了更好的选择，所以你不怎么需要它。

> "I know Futures, and they are completely useless!"

也许你知道些其他的 Future 实现，最引人注目的是 Java 提供的那个。 但是对于 Java 的 Future，你只能去查看它是否已经完成，或者阻塞线程直到其结束。 简而言之，Java 的 Future 几乎没有用，而且用起来绝对不会让人开心。

如果你认为 Scala 的 Future 也是这样，那大错特错了！

## Future 语义

*scala.concurrent* 包里的 `Future[T]` 是一个容器类型，代表一种返回值类型为 `T` 的计算。 计算可能会出错，也可能会超时；从而，当一个 future 完成时，它可能会包含异常，而不是你期望的那个值。

Future 只能写一次： 当一个 future 完成后，它就不能再被改变了。 同时，Future 只提供了读取计算值的接口，写入计算值的任务交给了 Promise，这样，API 层面上会有一个清晰的界限。 这篇文章里，我们主要关注前者，下一章会介绍 Promise 的使用。

## 使用 Future

Future 有多种使用方式，我将通过重写 “卡布奇诺” 这个例子来说明。

首先，所有可以并行执行的函数，应该返回一个 Future：

```
import scala.concurrent.future
import scala.concurrent.Future
import scala.concurrent.ExecutionContext.Implicits.global
import scala.concurrent.duration._
import scala.util.Random

def grind(beans: CoffeeBeans): Future[GroundCoffee] = Future {
  println("start grinding...")
  Thread.sleep(Random.nextInt(2000))
  if (beans == "baked beans") throw GrindingException("are you joking?")
  println("finished grinding...")
  s"ground coffee of $beans"
}

def heatWater(water: Water): Future[Water] = Future {
  println("heating the water now")
  Thread.sleep(Random.nextInt(2000))
  println("hot, it's hot!")
  water.copy(temperature = 85)
}

def frothMilk(milk: Milk): Future[FrothedMilk] = Future {
  println("milk frothing system engaged!")
  Thread.sleep(Random.nextInt(2000))
  println("shutting down milk frothing system")
  s"frothed $milk"
}

def brew(coffee: GroundCoffee, heatedWater: Water): Future[Espresso] = Future {
  println("happy brewing :)")
  Thread.sleep(Random.nextInt(2000))
  println("it's brewed!")
  "espresso"
}
```

上面的代码有几处需要解释。

首先是 Future 伴生对象里的 `apply` 方法需要两个参数：

```
object Future {
  def apply[T](body: => T)(implicit execctx: ExecutionContext):
Future[T]
}
```

要异步执行的计算通过传名参数 `body` 传入。 第二个参数是一个隐式参数，隐式参数是说，函数调用时，如果作用域中存在一个匹配的隐式值，就无需显示指定这个参数。 `ExecutionContext` 可以执行一个 Future，可以把它看作是一个线程池，是绝大部分 Future API 的隐式参数。

`import scala.concurrent.ExecutionContext.Implicits.global` 语句引入了一个全局的执行上下文，确保了隐式值的存在。 这时候，只需要一个单元素列表,可以用大括号来代替小括号。 调用 `future` 方法时，经常使用这种形式，使得它看起来像是一种语言特性，而不是一个普通方法的调用。

这个例子没有大量计算，所以用随机休眠来模拟以说明问题， 而且，为了更清晰的说明并发代码的执行顺序，还在“计算”之前和之后打印了些东西。

计算会在 Future 创建后的某个不确定时间点上由 `ExecutionContext` 给其分配的某个线程中执行。

## 回调

对于一些简单的问题，使用回调就能很好解决。 Future 的回调是偏函数，你可以把回调传递给 Future 的 `onSuccess` 方法， 如果这个 Future 成功完成，这个回调就会执行，并把 Future 的返回值作为参数输入：

```
grind("arabica beans").onSuccess { case ground =>
  println("okay, got my ground coffee")
}
```

类似的，也可以在 `onFailure` 上注册回调，只不过它是在 Future 失败时调用，其输入是一个 `Throwable` 。

通常的做法是将两个回调结合在一起以更好的处理 Future：在 `onComplete` 方法上注册回调，回调的输入是一个 Try。

```
import scala.util.{Success, Failure}
grind("baked beans").onComplete {
  case Success(ground) => println(s"got my $ground")
  case Failure(ex) => println("This grinder needs a replacement
, seriously!")
}
```

传递给 `grind` 的是 “baked beans”，因此 `grind` 方法会产生异常，进而导致 Future 中的计算失败。

## Future 组合

当嵌套使用 Future 时，回调就变得比较烦人。 不过，你也没必要这么做，因为 Future 是可组合的，这是它眞正发挥威力的时候！

你一定已经注意到，之前讨论过的所有容器类型都可以进行 `map` 、 `flatMap` 操作，也可以用在 for 语句中。 作为一种容器类型，Future 支持这些操作也不足为奇！

眞正的问题是，在还没有完成的计算上执行这些操作意味这什么，如何去理解它们？

## Map 操作

Scala 让 “时间旅行” 成为可能！ 假设想在水加热后就去检查它的温度， 可以通过将 `Future[Water]` 映射到 `Future[Boolean]` 来完成这件事情：

```
val tempreatureOkay: Future[Boolean] = heatWater(Water(25)) map
{ water =>
  println("we're in the future!")
  (80 to 85) contains (water.temperature)
}
```

`tempreatureOkay` 最终会包含水温的结果。 你可以去改变 `heatWater` 的实现来让它抛出异常（比如说，加热器爆炸了）， 然后等待 “we're in the future!” 出现在显示屏上，不过你永远等不到。

写传递给 `map` 的函数时，你就处在未来（或者说可能的未来）。 一旦 `Future[Water]` 实例成功完成，这个函数就会执行，只不过，该函数所在的时间线可能不是你现在所处的这个。 如果 `Future[Water]` 失败，传递给 `map` 的函数中的事情永远不会发生，调用 `map` 的结果将是一个失败的 `Future[Boolean]` 。

## FlatMap 操作

如果一个 Future 的计算依赖于另一个 Future 的结果，那需要求救于 `flatMap` 以避免 Future 的嵌套。

假设，测量水温的线程需要一些时间，那你可能想异步的去检查水温是否 OK。 比如，有一个函数，接受一个 `Water` ，并返回 `Future[Boolean]` ：

```
def temperatureOkay(water: Water): Future[Boolean] = future {
  (80 to 85) contains (water.temperature)
}
```

使用 `flatMap` （而不是 `map` ）得到一个 `Future[Boolean]` ，而不是 `Future[Future[Boolean]]` ：

```
val nestedFuture: Future[Future[Boolean]] = heatWater(Water(25))
 map {
  water => temperatureOkay(water)
}

val flatFuture: Future[Boolean] = heatWater(Water(25)) flatMap {
  water => temperatureOkay(water)
}
```

同样，映射只会发生在 `Future[Water]` 成功完成情况下。

## for 语句

除了调用 `flatMap` ，也可以写成 for 语句。上面的例子可以重写成：

```
val acceptable: Future[Boolean] = for {
  heatedWater <- heatWater(Water(25))
  okay <- temperatureOkay(heatedWater)
} yield okay
```

如果有多个可以并行执行的计算，则需要特别注意，要先在 for 语句外面创建好对应的 Futures。

```
def prepareCappuccinoSequentially(): Future[Cappuccino] =
  for {
    ground <- grind("arabica beans")
    water <- heatWater(Water(25))
    foam <- frothMilk("milk")
    espresso <- brew(ground, water)
  } yield combine(espresso, foam)
```

这看起来很漂亮，但要知道，for 语句只不过是 `flatMap` 嵌套调用的语法糖。这意味着，只有当 `Future[GroundCoffee]` 成功完成后，`heatWater` 才会创建 `Future[Water]`。你可以查看函数运行时打印出来的东西来验证这个说法。

因此，要确保在 for 语句之前实例化所有相互独立的 Futures：

```
def prepareCappuccino(): Future[Cappuccino] = {
  val groundCoffee = grind("arabica beans")
  val heatedWater = heatWater(Water(20))
  val frothedMilk = frothMilk("milk")
  for {
    ground <- groundCoffee
    water <- heatedWater
    foam <- frothedMilk
    espresso <- brew(ground, water)
  } yield combine(espresso, foam)
}
```

在 for 语句之前，三个 Future 在创建之后就开始各自独立的运行，显示屏的输出是不确定的。唯一能确定的是 “happy brewing” 总是出现在后面，因为该输出所在的函数 `brew` 是在其他两个函数执行完毕后才开始执行的。也因为此，可以在 for

语句里面直接调用它，当然，前提是前面的 Future 都成功完成。

## 失败偏向的 Future

你可能会发现 `Future[T]` 是成功偏向的，允许你使用 `map` 、 `flatMap` 、 `filter` 等。

但是，有时候可能处理事情出错的情况。 调用 `Future[T]` 上的 `failed` 方法，会得到一个失败偏向的 Future，类型是 `Future[Throwable]` 。 之后就可以映射这个 `Future[Throwable]` ，在失败的情况下执行 mapping 函数。

# 总结

你已经见过 Future 了，而且它的前途看起来很光明！ 因为它是一个可组合、可函数式使用的容器类型，这让我们的工作变得异常舒服。

调用 `future` 方法可以轻易将阻塞执行的代码变成并发执行，但是，代码最好原本就是非阻塞的。 为了实现它，我们还需要 `Promise` 来完成 `Future` ，这就是下一章的主题。

# 实战中的 Promise 和 Future

上一章介绍了 Future 类型，以及如何用它来编写高可读性、高组合性的异步执行代码。

Future 只是整个谜团的一部分： 它是一个只读类型，允许你使用它计算得到的值，或者处理计算中出现的错误。 但是在这之前，必须得有一种方法把这个值放进去。 这一章里，你将会看到如何通过 Promise 类型来达到这个目的。

## 类型 Promise

之前，我们把一段顺序执行的代码块传递给了 `scala.concurrent` 里的 `future` 方法， 并且在作用域中给出了一个 `ExecutionContext` ，它神奇地异步调用代码块，返回一个 Future 类型的结果。

虽然这种获得 Future 的方式很简单，但还有其他的方法来创建 Future 实例，并填充它，这就是 Promise。 Promise 允许你在 Future 里放入一个值，不过只能做一次，Future 一旦完成，就不能更改了。

一个 Future 实例总是和一个（也只能是一个）Promise 实例关联在一起。 如果你在 REPL 里调用 `future` 方法，你会发现返回的也是一个 Promise：

```
import concurrent.Future
import concurrent.Future

scala> import concurrent.future
import concurrent.future

scala> import concurrent.ExecutionContext.Implicits.global
import concurrent.ExecutionContext.Implicits.global

scala> val f: Future[String] = future { "Hello World!" }
f: scala.concurrent.Future[String] = scala.concurrent.impl.Promi
se$DefaultPromise@2b509249
```

你得到的对象是一个 `DefaultPromise` ，它实现了 `Future` 和 `Promise` 接口， 不过这就是具体的实现细节了（译注，有兴趣的读者可翻阅其实现的源码），使用者只需要知道代码实现把 Future 和对应的 Promise 之间的联系分的很清晰。

这个小例子说明了：除了通过 Promise，没有其他方法可以完成一个 Future， `future` 方法也只是一个辅助函数，隐藏了具体的实现机制。

现在，让我们动动手，看看怎样直接使用 Promise 类型。

## 给出承诺

当我们谈论起承诺能否被兑现时，一个很熟知的例子是那些政客的竞选诺言。

假设被推选的政客给他的投票者一个减税的承诺。 这可以用 `Promise[TaxCut]` 表示：

```
import concurrent.Promise
case class TaxCut(reduction: Int)
// either give the type as a type parameter to the factory metho
d:
val taxcut = Promise[TaxCut]()
// or give the compiler a hint by specifying the type of your va
l:
val taxcut2: Promise[TaxCut] = Promise()
// taxcut: scala.concurrent.Promise[TaxCut] = scala.concurrent.i
mpl.Promise$DefaultPromise@66ae2a84
// taxcut2: scala.concurrent.Promise[TaxCut] = scala.concurrent.
impl.Promise$DefaultPromise@346974c6
```

一旦创建了这个 Promise，就可以在它上面调用 `future` 方法来获取承诺的未来：

```
 val taxCutF: Future[TaxCut] = taxcut.future
 // `> scala.concurrent.Future[TaxCut] `  scala.concurrent.impl.
Promise$DefaultPromise@66ae2a84
```

返回的 Future 可能并不和 Promise 一样，但在同一个 Promise 上调用 `future` 方法总是返回同一个对象， 以确保 Promise 和 Future 之间一对一的关系。

## 结束承诺

一旦给出了承诺，并告诉全世界会在不远的将来兑现它，那最好尽力去实现。 在 Scala 中，可以结束一个 Promise，无论成功还是失败。

兑现承诺

为了成功结束一个 Promise，你可以调用它的 `success` 方法，并传递一个大家期许的结果：

```
taxcut.success(TaxCut(20))
```

这样做之后，Promise 就无法再写入其他值了，如果偏要再写，会产生异常。

此时，和 Promise 关联的 Future 也成功完成，注册的回调会开始执行， 或者说对这个 Future 进行了映射，那这个时候，映射函数也该执行了。

一般来说，Promise 的完成和对返回的 Future 的处理发生在不同的线程。 很可能你创建了 Promise，并立即返回和它关联的 Future 给调用者，而实际上，另外一个线程还在计算它。

为了说明这一点，我们拿减税来举个例子：

```
object Government {
  def redeemCampaignPledge(): Future[TaxCut] = {
    val p = Promise[TaxCut]()
    Future {
      println("Starting the new legislative period.")
      Thread.sleep(2000)
      p.success(TaxCut(20))
      println("We reduced the taxes! You must reelect us!!!!1111"
)
    }
    p.future
  }
}
```

这个例子中使用了 Future 伴生对象，不过不要被它搞混淆了，这个例子的重点是：Promise 并不是在调用者的线程里完成的。

现在我们来兑现当初的竞选宣言，在 Future 上添加一个 `onComplete` 回调：

```
import scala.util.{Success, Failure}
val taxCutF: Future[TaxCut] = Government.redeemCampaignPledge()
println("Now that they're elected, let's see if they remember their promises...")
taxCutF.onComplete {
  case Success(TaxCut(reduction)) =>
    println(s"A miracle! They really cut our taxes by $reduction percentage points!")
  case Failure(ex) =>
    println(s"They broke their promises! Again! Because of a ${ex.getMessage}")
}
```

多次运行这个例子，会发现显示屏输出的结果顺序是不确定的，而且，最终回调函数会执行，进入成功的那个 case 。

### 违背诺言

政客习惯违背诺言，Scala 程序员有时候也只能这样做。 调用 `failure` 方法，传递一个异常，结束 Promise：

```
case class LameExcuse(msg: String) extends Exception(msg)
object Government {
  def redeemCampaignPledge(): Future[TaxCut] = {
     val p = Promise[TaxCut]()
     Future {
       println("Starting the new legislative period.")
       Thread.sleep(2000)
       p.failure(LameExcuse("global economy crisis"))
       println("We didn't fulfill our promises, but surely they'
ll understand.")
     }
     p.future
   }
}
```

这个 `redeemCampaignPledge` 实现最终会违背承诺。 一旦用 `failure` 结束这个 Promise，也无法再次写入了，正如 `success` 方法一样。 相关联的 Future 也会以 `Failure` 收场。

如果已经有了一个 Try，那可以直接把它传递给 Promise 的 `complete` 方法，以此来结束这个它。 如果这个 Try 是一个 Success，关联的 Future 会成功完成，否则，就失败。

## 基于 Future 的编程实践

如果想使用基于 Future 的编程范式以增加应用的扩展性，那应用从下到上都必须被设计成非阻塞模式。 这意味着，基本上应用层所有的函数都应该是异步的，并且返回 Future。

当下，一个可能的使用场景是开发 Web 应用。 流行的 Scala Web 框架，允许你将响应作为 `Future[Response]` 返回，而不是等到你完成响应再返回。 这个非常重要，因为它允许 Web 服务器用少量的线程处理更多的连接。 通过赋予服务器 `Future[Response]` 的能力，你可以最大化服务器线程池的利用率。

而且，应用的服务可能需要多次调用数据库层以及（或者）某些外部服务， 这时候可以获取多个 Future，用 for 语句将它们组合成新的 Future，简单可读！ 最终，Web 层再将这样的一个 Future 变成 `Future[Response]` 。

但是该怎样在实践中实现这些呢？需要考虑三种不同的场景：

## 非阻塞IO

应用很可能涉及到大量的 IO 操作。比如，可能需要和数据库交互，还可能作为客户端去调用其他的 Web 服务。

如果是这样，可以使用一些基于 Java 非阻塞 IO 实现的库，也可以直接或通过 Netty 这样的库来使用 Java 的 NIO API。 这样的库可以用定量的线程池处理大量的连接。

但如果是想开发这样的一个库，直接和 Promise 打交道更为合适。

## 阻塞 IO

有时候，并没有基于 NIO 的库可用。比如，Java 世界里大多数的数据库驱动都是使用阻塞 IO。 在 Web 应用中，如果用这样的驱动发起大量访问数据库的调用，要记得这些调用是发生在服务器线程里的。 为了避免这个问题，可以将所有需要和数据库交互的代码都放入 `future` 代码块里，就像这样：

```
// get back a Future[ResultSet] or something similar:
Future {
  queryDB(query)
}
```

到现在为止，我们都是使用隐式可用的全局 `ExecutionContext` 来执行这些代码块。 通常，更好的方式是创建一个专用的 `ExecutionContext` 放在数据库层里。 可以从 Java的 `ExecutorService` 来它，这也意味着，可以异步的调整线程池来执行数据库调用，应用的其他部分不受影响。

```
import java.util.concurrent.Executors
import concurrent.ExecutionContext
val executorService = Executors.newFixedThreadPool(4)
val executionContext = ExecutionContext.fromExecutorService(exec
utorService)
```

## 长时间运行的计算

取决于应用的本质特点，一个应用偶尔还会调用一些长时间运行的任务，它们完全不涉及 IO（CPU 密集的任务）。 这些任务也不应该在服务器线程中执行，因此需要将它们变成 Future：

```
Future {
  longRunningComputation(data, moreData)
}
```

同样，最好有一些专属的 `ExecutionContext` 来处理这些 CPU 密集的计算。 怎样调整这些线程池大小取决于应用的特征，这些已经超过了本文的范围。

## 总结

这一章里，我们学习了 Promise - 基于 Future 的并发范式的可写组件，以及怎样用它来完成一个 Future； 同时，还给出了一些在实践中使用它们的建议。

下一章会讨论 Scala 函数式编程是如何增加代码可用性（一个长久以来和面向对象编程相关联的概念）的。

# 高阶函数与 DRY

前几章介绍了 Scala 容器类型的可组合性特征。 接下来，你会发现，Scala 中的一等公民——函数也具有这一性质。

组合性产生可重用性，虽然后者是经由面向对象编程而为人熟知，但它也绝对是纯函数的固有性质。 （纯函数是指那些没有副作用且是引用透明的函数）

一个明显的例子是调用已知函数实现一个新的函数，当然，还有其他的方式来重用已知函数。 这一章会讨论函数式编程的一些基本原理。 你将会学到如何使用高阶函数，以及重用已有代码时，遵守 DRY 原则。

## 高阶函数

和一阶函数相比，高阶函数可以有三种形式：

1. 一个或多个参数是函数，并返回一个值。
2. 返回一个函数，但没有参数是函数。
3. 上述两者叠加：一个或多个参数是函数，并返回一个函数。

看到这里的读者应该已经见到过第一种使用：我们调用一个方法，像 `map` 、 `filter` 、 `flatMap` ，并传递另一个函数给它。 传递给方法的函数通常是匿名函数，有时候，还涉及一些代码冗余。

这一章只关注另外两种功能：一个可以根据输入值构建新的函数，另一个可以根据现有的函数组合出新的函数。 这两种情况都能够消除代码冗余。

## 函数生成

你可能认为依据输入值创建新函数的能力并不是那么有用。 函数组合非常重要，但在这之前，还是先来看看如何使用可以产生新函数的函数。

假设要实现一个免费的邮件服务，用户可以设置对邮件的屏蔽。 我们用一个简单的样例类来代表邮件：

```
case class Email(
  subject: String,
  text: String,
  sender: String,
  recipient: String
)
```

想让用户可以自定义过滤条件，需有一个过滤函数——类型为 `Email => Boolean` 的谓词函数， 这个谓词函数决定某个邮件是否该被屏蔽：如果谓词成真，那这个邮件被接受，否则就被屏蔽掉。

```
type EmailFilter = Email => Boolean
def newMailsForUser(mails: Seq[Email], f: EmailFilter) = mails.f
ilter(f)
```

注意，类型别名使得代码看起来更有意义。

现在，为了使用户能够配置邮件过滤器，实现了一些可以产生 `EmailFilter` 的工厂方法：

```
  val sentByOneOf: Set[String] => EmailFilter =
    senders =>
      email => senders.contains(email.sender)
  val notSentByAnyOf: Set[String] => EmailFilter =
    senders =>
      email => !senders.contains(email.sender)
  val minimumSize: Int => EmailFilter =
    n =>
      email => email.text.size >= n
  val maximumSize: Int => EmailFilter =
    n =>
      email => email.text.size <= n
```

这四个 *vals* 都是可以返回 `EmailFilter` 的函数， 前两个接受代表发送者的 `Set[String]` 作为输入，后两个接受代表邮件内容长度的 `Int` 作为输入。

可以使用这些函数来创建 `EmailFilter` ：

```
  val emailFilter: EmailFilter = notSentByAnyOf(Set("johndoe@example.com"))
  val mails = Email(
    subject = "It's me again, your stalker friend!",
    text = "Hello my friend! How are you?",
    sender = "johndoe@example.com",
    recipient = "me@example.com") :: Nil
  newMailsForUser(mails, emailFilter) // returns an empty list
```

这个过滤器过滤掉列表里唯一的一个元素，因为用户屏蔽了来自 `johndoe@example.com` 的邮件。 可以用工厂方法创建任意的 `EmailFilter` 函数，这取决于用户的需求了。

## 重用已有函数

当前的解决方案有两个问题。第一个是工厂方法中有重复代码。 上文提到过，函数的组合特征可以很轻易的保持 DRY 原则，既然如此，那就试着使用它吧！

对于 `minimumSize` 和 `maximumSize` ，我们引入一个叫做 `sizeConstraint` 的函数。 这个函数接受一个谓词函数，该谓词函数检查函数内容长度是否OK，邮件长度会通过参数传递给它：

```
  type SizeChecker = Int => Boolean
  val sizeConstraint: SizeChecker => EmailFilter =
    f =>
      email => f(email.text.size)
```

这样，我们就可以用 `sizeConstraint` 来表示 `minimumSize` 和 `maximumSize` 了：

```
  val minimumSize: Int => EmailFilter =
    n =>
      sizeConstraint(_ >= n)
  val maximumSize: Int => EmailFilter =
    n =>
      sizeConstraint(_ <= n)
```

# 函数组合

为另外两个谓词（`sentByOneOf` 、 `notSentByAnyOf` ）介绍一个通用的高阶函数，通过它，可以用一个函数去表达另外一个函数。

这个高阶函数就是 `complement` ，给定一个类型为 `A => Boolean` 的谓词，它返回一个新函数， 这个新函数总是得出和谓词相对立的结果：

```
def complement[A](predicate: A => Boolean) = (a: A) => !predicate(a)
```

现在，对于一个已有的谓词 `p` ，调用 `complement(p)` 可以得到它的补。 然而， `sentByAnyOf` 并不是一个谓词函数，它返回类型为 `EmailFilter` 的谓词。

Scala 函数的可组合能力现在就用的上了：给定两个函数 `f` 、 `g` ， `f.compose(g)` 返回一个新函数， 调用这个新函数时，会首先调用 `g` ，然后应用 `f` 到 `g` 的返回结果上。 类似的， `f.andThen(g)` 返回的新函数会应用 `g` 到 `f` 的返回结果上。

知道了这些，我们就可以重写 `notSentByAnyOf` 了：

```
val notSentByAnyOf = sentByOneOf andThen (g => complement(g))
```

上面的代码创建了一个新的函数， 这个函数首先应用 `sentByOneOf` 到参数 `Set[String]` 上，产生一个 `EmailFilter` 谓词, 然后，应用 `complement` 到这个谓词上。 使用 Scala 的下划线语法，这短代码还能更精简：

```
val notSentByAnyOf = sentByOneOf andThen (complement(_))
```

读者可能已经注意到， 给定 `complement` 函数，也可以通过 `minimumSize` 来实现 `maximumSize` 。 不过，先前的实现方式更加灵活，它允许检查邮件内容的任意长度。 谓

## 谓词组合

邮件过滤器的第二个问题是，当前只能传递一个 `EmailFilter` 给 `newMailsForUser` 函数，而用户必然想设置多个标准。 所以需要可以一种可以创建组合谓词的方法，这个组合谓词可以在任意一个标准满足的情况下返回 `true` ，或者在都不满足时返回 `false` 。

下面的代码是一种实现方式：

```
  def any[A](predicates: (A => Boolean)*): A => Boolean =
    a => predicates.exists(pred => pred(a))
  def none[A](predicates: (A => Boolean)*) = complement(any(pred
icates: _*))
  def every[A](predicates: (A => Boolean)*) = none(predicates.vi
ew.map(complement(_)): _*)
```

`any` 函数返回的新函数会检查是否有一个谓词对于输入 `a` 成真。 `none` 返回的是 `any` 返回函数的补，只要存在一个成真的谓词， `none` 的条件就无法满足。 最后， `every` 利用 `none` 和 `any` 来判定是否每个谓词的补对于输入 `a` 都不成真。

可以使用它们来创建代表用户设置的组合 `EmailFilter` ：

```
  val filter: EmailFilter = every(
      notSentByAnyOf(Set("johndoe@example.com")),
      minimumSize(100),
      maximumSize(10000)
    )
```

## 流水线组合

再举一个函数组合的例子。回顾下上面的场景， 邮件提供者不仅想让用户可以配置邮件过滤器，还想对用户发送的邮件做一些处理。 这是一些简单的 `Email => Email` 函数，一些可能的处理函数是：

```
  val addMissingSubject = (email: Email) =>
    if (email.subject.isEmpty) email.copy(subject = "No subject")
    else email
  val checkSpelling = (email: Email) =>
    email.copy(text = email.text.replaceAll("your", "you're"))
  val removeInappropriateLanguage = (email: Email) =>
    email.copy(text = email.text.replaceAll("dynamic typing", "**CENSORED**"))
  val addAdvertismentToFooter = (email: Email) =>
    email.copy(text = email.text + "\nThis mail sent via Super Awesome Free Mail")
```

现在，根据老板的心情，可以按需配置邮件处理的流水线。 通过 `andThen` 调用实现，或者使用 Function 伴生对象上的 `chain` 方法：

```
  val pipeline = Function.chain(Seq(
    addMissingSubject,
    checkSpelling,
    removeInappropriateLanguage,
    addAdvertismentToFooter))
```

# 高阶函数与偏函数

这部分不会关注细节，不过，在知道了这么多通过高阶函数来组合和重用函数的方法之后，你可能想再重新看看偏函数。

## 链接偏函数

匿名函数那一章提到过，偏函数可以被用来创建责任链： `PartialFunction` 上的 `orElse` 方法允许链接任意个偏函数，从而组合出一个新的偏函数。 不过，只有在一个偏函数没有为给定输入定义的时候，才会把责任传递给下一个偏函数。 从而可以做下面这样的事情：

```
  val handler = fooHandler orElse barHandler orElse bazHandler
```

## 再看偏函数

有时候，偏函数并不合适。 仔细想想，一个函数没有为所有的输入值定义操作，这样的事实还可以用一个返回 `Option[A]` 的标准函数代替： 如果函数为一个输入定义了操作，那就返回 `Some[A]` ，否则返回 `None` 。

要这么做的话，可以在给定的偏函数 `pf` 上调用 `lift` 方法得到一个普通的函数，这个函数返回 `Option` 。 反过来，如果有一个返回 `Option` 的普通函数 `f` ，也可以调用 `Function.unlift(f)` 来得到一个偏函数。 总

# 总结

这一章给出了高阶函数的使用，利用它可以在一个新的环境里重用已有函数，并用灵活的方式去组合它们。 在所举的例子中，就代码行数而言，可能看不出太多价值， 这些例子都很简单，只是为了说明而已，在架构层面，组合和重用函数是有很大帮助的。

下一章，我们继续探索函数组合的方式：函数部分应用和柯里化*(Partial Function Application and Currying)*。

# 柯里化和部分函数应用

上一章重点在于代码重复：提升现有的函数功能、或者将函数进行组合。 这一章，我们来看看另外两种函数重用的机制：函数的部分应用**(Partial Application of Functions)** 、 柯里化**(Currying)** 。

## 部分应用的函数

和其他遵循函数式编程范式的语言一样，Scala 允许部分应用一个函数。 调用一个函数时，不是把函数需要的所有参数都传递给它，而是仅仅传递一部分，其他参数留空； 这样会生成一个新的函数，其参数列表由那些被留空的参数组成。（不要把这个概念和偏函数混淆）

为了具体说明这一概念，回到上一章的例子： 假想的免费邮件服务，能够让用户配置筛选器，以使得满足特定条件的邮件显示在收件箱里，其他的被过滤掉。

`Email` 类看起来仍然是这样：

```
case class Email(
  subject: String,
  text: String,
  sender: String,
  recipient: String)
type EmailFilter = Email => Boolean
```

过滤邮件的条件用谓词 `Email => Boolean` 表示， `EmailFilter` 是其别名。 调用适当的工厂方法可以生成这些谓词。

上一章，我们创建了两个这样的工厂方法，它们检查邮件内容长度是否满足给定的最大值或最小值。 这一次，我们使用部分应用函数来实现这些工厂方法，做法是，修改 `sizeConstraint` ，固定某些参数可以创建更具体的限制条件：

其修改后的代码如下：

```
    type IntPairPred = (Int, Int) => Boolean
    def sizeConstraint(pred: IntPairPred, n: Int, email: Email)
 =
      pred(email.text.size, n)
```

上述代码为一个谓词函数定义了别名 `IntPairPred` ，该函数接受一对整数（值 `n` 和邮件内容长度），检查邮件长度对于 `n` 是否 OK。

请注意，不像上一章的 `sizeConstraint` ，这一个并不返回新的 `EmailFilter` ，它只是简单的用参数做计算，返回一个布尔值。秘诀在于，你可以部分应用这个 `sizeConstraint` 来得到一个 `EmailFilter` 。

遵循 DRY 原则，我们先来定义常用的 `IntPairPred` 实例，这样，在调用 `sizeConstraint` 时，不需要重复的写相同的匿名函数，只需传递下面这些：

```
    val gt: IntPairPred = _ > _
    val ge: IntPairPred = _ >= _
    val lt: IntPairPred = _ < _
    val le: IntPairPred = _ <= _
    val eq: IntPairPred = _ == _
```

最后，调用 `sizeConstraint` 函数，用上面的 `IntPairPred` 传入第一个参数：

```
    val minimumSize: (Int, Email) => Boolean = sizeConstraint(ge
, _: Int, _: Email)
    val maximumSize: (Int, Email) => Boolean = sizeConstraint(le
, _: Int, _: Email)
```

对所有没有传入值的参数，必须使用占位符 `_` ，还需要指定这些参数的类型，这使得函数的部分应用多少有些繁琐。Scala 编译器无法推断它们的类型，方法重载使编译器不可能知道你想使用哪个方法。

不过，你可以绑定或漏掉任意个、任意位置的参数。比如，我们可以漏掉第一个值，只传递约束值 `n` ：

```
    val constr20: (IntPairPred, Email) => Boolean =
      sizeConstraint(_: IntPairPred, 20, _: Email)

    val constr30: (IntPairPred, Email) => Boolean =
      sizeConstraint(_: IntPairPred, 30, _: Email)
```

得到的两个函数，接受一个 `IntPairPred` 和一个 `Email` 作为参数，然后利用谓词函数 `IntPairPred` 把邮件长度和 `20` 、 `30` 比较，只不过比较方法的逻辑 `IntPairPred` 需要另外指定。

由此可见，虽然函数部分应用看起来比较冗长，但它要比 Clojure 的灵活，在 Clojure 里，必须从左到右的传递参数，不能略掉中间的任何参数。

## 从方法到函数对象

在一个方法上做部分应用时，可以不绑定任何的参数，这样做的效果是产生一个函数对象，并且其参数列表和原方法一模一样。通过这种方式可以将方法变成一个可赋值、可传递的函数！

```
    val sizeConstraintFn: (IntPairPred, Int, Email) => Boolean =
  sizeConstraint _
```

# 更有趣的函数

部分函数应用显得太啰嗦，用起来不够优雅，幸好还有其他的替代方法。

也许你已经知道 Scala 里的方法可以有多个参数列表。下面的代码用多个参数列表重新定义了 `sizeConstraint` ：

```
    def sizeConstraint(pred: IntPairPred)(n: Int)(email: Email):
 Boolean =
      pred(email.text.size, n)
```

如果把它变成一个可赋值、可传递的函数对象，它的签名看起来会像是这样：

```
val sizeConstraintFn: IntPairPred => Int => Email => Boolean
= sizeConstraint _
```

这种单参数的链式函数称做 柯里化函数 ，以发明人 Haskell Curry 命名。在 Haskell 编程语言里，所有的函数默认都是柯里化的。

`sizeConstraintFn` 接受一个 `IntPairPred` ，返回一个函数，这个函数又接受 `Int` 类型的参数，返回另一个函数，最终的这个函数接受一个 `Email` ，返回布尔值。

现在，可以把要传入的 `IntPairPred` 传递给 `sizeConstraint` 得到：

```
val minSize: Int => Email => Boolean = sizeConstraint(ge)
val maxSize: Int => Email => Boolean = sizeConstraint(le)
```

被留空的参数没必要使用占位符，因为这不是部分函数应用。

现在，可以通过这两个柯里化函数来创建 `EmailFilter` 谓词：

```
val min20: Email => Boolean = minSize(20)
val max20: Email => Boolean = maxSize(20)
```

也可以在柯里化的函数上一次性绑定多个参数，直接得到上面的结果。 传入第一个参数得到的函数会立即应用到第二个参数上：

```
val min20: Email => Boolean = sizeConstraintFn(ge)(20)
val max20: Email => Boolean = sizeConstraintFn(le)(20)
```

## 函数柯里化

有时候，并不总是能提前知道要不要将一个函数写成柯里化形式，毕竟，和只有单参数列表的函数相比，柯里化函数的使用并不清晰。 而且，偶尔还会想以柯里化的形式去使用第三方的函数，但这些函数的参数都在一个参数列表里。

这就需要一种方法能对函数进行柯里化。这种的柯里化行为本质上也是一个高阶函数：接受现有的函数，返回新函数。这个高阶函数就是 `curried` ：`curried` 方法存在于 `Function2` 、 `Function3` 这样的多参数函数类型里。如果存在一个接受两个参数的 `sum` ，可以通过调用 `curried` 方法得到它的柯里化版本：

```
val sum: (Int, Int) => Int = _ + _
val sumCurried: Int => Int => Int = sum.curried
```

使用 `Funtion.uncurried` 进行反向操作，可以将一个柯里化函数转换成非柯里化版本。

## 函数化的依赖注入

在这一章的最后，我们来看看柯里化函数如何发挥其更大的作用。来自 Java 或者 .NET 世界的人，或多或少都用过依赖注入容器，这些容器为使用者管理对象，以及对象之间的依赖关系。在 Scala 里，你并不真的需要这样的外部工具，语言已经提供了许多功能，这些功能简化了依赖注入的实现。

函数式编程仍然需要注入依赖：应用程序中上层函数需要调用其他函数。把要调用的函数硬编码在上层函数里，不利于它们的独立测试。从而需要把被依赖的函数以参数的形式传递给上层函数。

但是，每次调用都传递相同的依赖，是不符合 DRY 原则的，这时候，柯里化函数就有用了！柯里化和部分函数应用是函数式编程里依赖注入的几种方式之一。

下面这个简化的例子说明了这项技术：

```
case class User(name: String)
trait EmailRepository {
  def getMails(user: User, unread: Boolean): Seq[Email]
}
trait FilterRepository {
  def getEmailFilter(user: User): EmailFilter
}
trait MailboxService {
  def getNewMails(emailRepo: EmailRepository)(filterRepo: FilterRepository)(user: User) =
    emailRepo.getMails(user, true).filter(filterRepo.getEmailFilter(user))
  val newMails: User => Seq[Email]
}
```

这个例子有一个依赖两个不同存储库的服务，这些依赖被声明为 `getNewMails` 方法的参数，并且每个依赖都在一个单独的参数列表里。

`MailboxService` 实现了这个方法，留空了字段 `newMails` ，这个字段的类型是一个函数： `User => Seq[Email]` ，依赖于 `MailboxService` 的组件会调用这个函数。

扩展 `MailboxService` 时，实现 `newMails` 的方法就是应用 `getNewMails` 这个方法，把依赖 `EmailRepository` 、 `FilterRepository` 的具体实现传递给它：

```
object MockEmailRepository extends EmailRepository {
  def getMails(user: User, unread: Boolean): Seq[Email] = Nil

}
object MockFilterRepository extends FilterRepository {
  def getEmailFilter(user: User): EmailFilter = _ => true
}
object MailboxServiceWithMockDeps extends MailboxService {
  val newMails: (User) => Seq[Email] =
    getNewMails(MockEmailRepository)(MockFilterRepository) _
}
```

调用 `MailboxServiceWithMockDeps.newMails(User("daniel")` 无需指定要使用的存储库。 在实际的应用程序中，这个服务也可能是以依赖的方式被使用，而不是直接引用。

这可能不是最强大、可扩展的依赖注入实现方式，但依旧是一个非常不错的选择，对展示部分函数应用和柯里化更广泛的功用来说，这也是一个不错的例子。 如果你想知道更多关于这一点的知识，推荐看 Debasish Ghosh 的幻灯片 “Dependency Injection in Scala”。

# 总结

这一章讨论了两个附加的可以避免代码重复的函数式编程技术， 并且在这个基础上，得到了很大的灵活性，可以用多种不同的形式重用函数。 部分函数应用和柯里化，这两者或多或少都可以实现同样的效果，只是有时候，其中的某一个会更为优雅。 下一章会继续探讨保持灵活性的方法：类型类（type class）。

# 类型类

前两章讨论了几种保持 DRY 和灵活性的函数式编程技术：

1. 函数组合（function composition）
2. 部分函数应用（partial function application）
3. 柯里化（currying）

这一章依旧围绕代码灵活性而来，不过不再讨论作为头等公民的函数，而是类型系统（注意：并不是要真的去研究类型系统）。 你将学习 类型类 ！

可能你会觉得这没有实际意义，认为这是被 Haskell 狂热分子带入 Scala 社区的异国情调，显然不是这样。 类型类已经成为 Scala 标准库，甚至是很多流行的、广泛使用的第三方开源库的重要组成部分，了解和熟悉类型类是很有必要的。

本章会讨论：

1. 类型类的概念，
2. 它为什么有用，
3. 使用它如何受益，
4. 如何实现类型类，并用于实践。

## 问题

我们用例子，而不是一个对类型类的抽象解释，开始本文的主题，例子简化了概念，也相当实用。

假设想提供一系列可以操作数字集合的函数，主要是计算它们的聚合值。 进一步假设只能通过索引来访问集合的元素，只能使用定义在 Scala 集合上的 `reduce` 方法。 （施加这些限制，是因为要实现的东西，Scala 标准库已经提供了） 最后，假定得到的值已排序。

我们先从 `median` ， `quartiles` ， `iqr` 的一个粗暴实现开始：

```
object Statistics {
  def median(xs: Vector[Double]): Double = xs(xs.size / 2)
  def quartiles(xs: Vector[Double]): (Double, Double, Double
) =
    (xs(xs.size / 4), median(xs), xs(xs.size / 4 * 3))
  def iqr(xs: Vector[Double]): Double = quartiles(xs) match
{
    case (lowerQuartile, _, upperQuartile) => upperQuartile
- lowerQuartile
  }
  def mean(xs: Vector[Double]): Double = {
    xs.reduce(_ + _) / xs.size
  }
}
```

`median` 将数据集分成两半，下四分位数和上四分位数（ `quartiles` 方法返回的元组的第一、第三个元素）分别分割了数据集的 25% 。 `iqr` 方法返回四分差（上四分卫数和下四分位数的差）。

现在我们想支持更多的类型，比如， `Int` ，所以应该为这个类型实现上面这些方法，对吧？

不！不能想当然的为 `Vector[Int]` 重载上面的方法（诡异的技巧除外），因为类型参数会被擦除，而且这样做有代码冗余的嫌疑。

要是 `Int` 和 `Double` 扩展自一个共同的基类，或者都实现了一个像是 `Number` 这样的特质，那该多好！

你可能会想着去把上述方法需要的参数类型替换成更通用的类型，看起来会是这样：

```
object Statistics {
  def median(xs: Vector[Number]): Number = ???
  def quartiles(xs: Vector[Number]): (Number, Number, Number
) = ???
  def iqr(xs: Vector[Number]): Number = ???
  def mean(xs: Vector[Number]): Number = ???
}
```

这样做，不仅丢掉了先前的类型信息，还违背了扩展性：不能强制第三方的数字类型扩展 `Number` 特质。 幸运的是，本例并不存在这样一个通用的特质。

对于这种问题，Ruby 的做法是 猴子补丁（**monkey patching**） ，扩展新类型让它看起来像一个 `Number` ，但是这样会污染全局命名空间。 年轻时遭到“四人帮”打击的 Java 开发者，则会认为 适配器（**Adpater**） 能解决上面所有问题：

> “四人帮”这里指的是设计模式一书的作者：Erich Gamma、Richard Helm、Ralph Johnson 和 John Vlissides， 具体见：http://en.wikipedia.org/wiki/Design_Patterns

```
    object Statistics {
      trait NumberLike[A] {
        def get: A
        def plus(y: NumberLike[A]): NumberLike[A]
        def minus(y: NumberLike[A]): NumberLike[A]
        def divide(y: Int): NumberLike[A]
      }
      case class NumberLikeDouble(x: Double) extends NumberLike[
Double] {
        def get: Double = x
        def minus(y: NumberLike[Double]) = NumberLikeDouble(x -
y.get)
        def plus(y: NumberLike[Double]) = NumberLikeDouble(x + y
.get)
        def divide(y: Int) = NumberLikeDouble(x / y)
      }
      type Quartile[A] = (NumberLike[A], NumberLike[A], NumberLi
ke[A])
      def median[A](xs: Vector[NumberLike[A]]): NumberLike[A] =
xs(xs.size / 2)
      def quartiles[A](xs: Vector[NumberLike[A]]): Quartile[A] =
        (xs(xs.size / 4), median(xs), xs(xs.size / 4 * 3))
      def iqr[A](xs: Vector[NumberLike[A]]): NumberLike[A] = qua
rtiles(xs) match {
        case (lowerQuartile, _, upperQuartile) => upperQuartile.
minus(lowerQuartile)
      }
      def mean[A](xs: Vector[NumberLike[A]]): NumberLike[A] =
        xs.reduce(_.plus(_)).divide(xs.size)
    }
```

上述代码解决了扩展性问题：使用这个库的用户可以将类型通过 `NumberLike` 适配器传递过来，无需重新编译统计库。

但是，把数字封装在适配器里，这样的代码会令人厌倦，无论读写，而且和统计库交互时，必须创建一大堆适配器实例。

## 类型类来救援

对目前所介绍的方法来说，类型类是一个强大的替代。 类型类是 Haskell 语言一个突出的特征，虽然它的名字里有类，但它和面向对象编程里的类没有任何关系。

一个类型类 `C` 定义了一些行为，要想成为 `C` 的一员，类型 `T` 必须支持这些行为。 一个类型 `T` 到底是不是 类型类 `C` 的成员，这一点并不是与生俱来的。 开发者可以实现类必须支持的行为，使得这个类变成类型类的成员。 一旦 `T` 变成 类型类 `C` 的一员，参数类型为类型类 `C` 成员的函数就可以接受类型 `T` 的实例。

这样，类型类支持临时的、追溯性的多态，依赖类型类的代码支持扩展性，且无需创建任何适配器对象。

## 创建类型类

Scala 中，类型类可以通过技术组合来实现和使用，比之 Haskell，它在 Scala 里的参与度更高，而且带给开发者更多的控制。

创建一个类型类涉及到几个步骤。

首先，我们来定义一个特质：

```
object Math {
  trait NumberLike[T] {
    def plus(x: T, y: T): T
    def divide(x: T, y: Int): T
    def minus(x: T, y: T): T
  }
}
```

上述代码创建了名为 `NumberLike` 的类型类特质。 类型类总会带着一个或多个类型参数，通常是无状态的，比如：里面定义的方法只对传入的参数进行操作。 前文的适配器操作的是它自己的字段和接受的一个参数，而这里定义的方法都需要两个参数，其中第一个参数对应适配器中的字段。

## 提供默认成员

第二步通常是在伴生对象里提供一些默认的类型类特质实现，之后你会知道为什么要这么做。 在这之前，先来实现 `Double` 和 `Int` 的类型类特质：

```
object Math {
  trait NumberLike[T] {
    def plus(x: T, y: T): T
    def divide(x: T, y: Int): T
    def minus(x: T, y: T): T
  }
  object NumberLike {
    implicit object NumberLikeDouble extends NumberLike[Doub
le] {
      def plus(x: Double, y: Double): Double = x + y
      def divide(x: Double, y: Int): Double = x / y
      def minus(x: Double, y: Double): Double = x - y
    }
    implicit object NumberLikeInt extends NumberLike[Int] {
      def plus(x: Int, y: Int): Int = x + y
      def divide(x: Int, y: Int): Int = x / y
      def minus(x: Int, y: Int): Int = x - y
    }
  }
}
```

两件事情： 第一，这两个实现基本相同。但不总是这样，毕竟 `NumberLike` 只是一个很小的域。 后面会给出类型类的一些例子，当为这些例子实现多个类型时，重复的余地就少很多。 第二， `NumberLikeInt` 做整数除法的时候，会损失一些精度，请忽略这一事实，这只是为简单起见。

你也许会发现，类型类的成员通常是单例对象，而且会有一个 `implicit` 关键字位于前面， 这是类型类在 Scala 中成为可能的几个重要因素之一，在某些条件下，它让类型类成员隐式可用。 更多相关的知识在下一节。

## 运用类型类

有了类型类和两个默认实现之后，就可以根据它们来实现统计。 我们先将重点放在 `mean` 方法上：

```
    object Statistics {
      import Math.NumberLike
      def mean[T](xs: Vector[T])(implicit ev: NumberLike[T]): T
 =
        ev.divide(xs.reduce(ev.plus(_, _)), xs.size)
    }
```

这样的代码初看起来可能有点吓人，实际上是相当简单，方法带有一个类型参数 `T` ，接受类型为 `Vector[T]` 的参数。

将参数限制在特定类型类的成员上，是通过第二个 `implicit` 参数列表实现的。这是什么意思？这是说，当前作用域中必须存在一个隐式可用的 `NumberLike[T]` 对象，比如说，当前作用域声明了一个 隐式值**(implicit value)**。 这种声明很多时候都是通过导入一个有隐式值定义的包或者对象来实现的。

当且仅当没有发现其他隐式值时，编译器会在隐式参数类型的伴生对象中寻找。 作为库的设计者，将默认的类型类实现放在伴生对象里意味着库的使用者可以轻易的重写默认实现，这正是库设计者喜闻乐见的。 用户还可以为隐式参数传递一个显示值，来重写作用域内的隐式值。

让我们来验证下默认的实现是否可以被正确解析：

```
    val numbers = Vector[Double](13, 23.0, 42, 45, 61, 73, 96, 1
 00, 199, 420, 900, 3839)
    println(Statistics.mean(numbers))
```

漂亮极了！试试 `Vector[String]` ，你会在编译期得到一个错误，这个错误指出参数 `ev: NumberLike[String]` 没有隐式值可用。 如果你不喜欢这个错误消息，你可以用 `@implicitNotFound` 为类型类添加批注，来自定义错误消息：

```
object Math {
  import annotation.implicitNotFound
  @implicitNotFound("No member of type class NumberLike in scope for ${T}")
  trait NumberLike[T] {
    def plus(x: T, y: T): T
    def divide(x: T, y: Int): T
    def minus(x: T, y: T): T
  }
}
```

## 上下文绑定

总是带着这个隐式参数列表显得有些冗长。 对于只有一个类型参数的隐式参数，Scala 提供了一种叫做 上下文绑定**(context bound)** 的简写。 为了说明这一使用方法，我们用它来实现剩下的统计方法：

```
object Statistics {
  import Math.NumberLike
  def mean[T](xs: Vector[T])(implicit ev: NumberLike[T]): T =
    ev.divide(xs.reduce(ev.plus(_, _)), xs.size)
  def median[T : NumberLike](xs: Vector[T]): T = xs(xs.size / 2)
  def quartiles[T: NumberLike](xs: Vector[T]): (T, T, T) =
    (xs(xs.size / 4), median(xs), xs(xs.size / 4 * 3))
  def iqr[T: NumberLike](xs: Vector[T]): T = quartiles(xs) match {
    case (lowerQuartile, _, upperQuartile) =>
      implicitly[NumberLike[T]].minus(upperQuartile, lowerQuartile)
  }
}
```

上下文绑定 `T: NumberLike` 意思是，必须有一个类型为 `NumberLike[T]` 的隐式值在当前上下文中可用，这和隐式参数列表是等价的。 如果想要访问这个隐式值，需要调用 `implicitly` 方法，就像上述 `iqr` 方法所做的那样。 如果类型

类需要多个类型参数，就不能使用上下文绑定语法了。

## 自定义的类型类成员

含有类型类的库的使用者，或迟或早会想将他自己的类型加入到类型类成员中。 比如说，可能想将统计用在 Joda Time 的 `Duration` 实例上。

我们来试试吧。首先将 Joda Time 加入到路径里：

```
libraryDependencies += "joda-time" % "joda-time" % "2.1"

libraryDependencies += "org.joda" % "joda-convert" % "1.3"
```

现在，只需创建 `NumberLike` 的一个隐式实现：

```
object JodaImplicits {
  import Math.NumberLike
  import org.joda.time.Duration
  implicit object NumberLikeDuration extends NumberLike[Duration] {
    def plus(x: Duration, y: Duration): Duration = x.plus(y)
    def divide(x: Duration, y: Int): Duration = Duration.millis(x.getMillis / y)
    def minus(x: Duration, y: Duration): Duration = x.minus(y)
  }
}
```

导入包含这个实现的包或者对象，就可以计算一堆 durations 的平均值了：

```
    import Statistics._
    import JodaImplicits._
    import org.joda.time.Duration._

    val durations = Vector(standardSeconds(20), standardSeconds(
57), standardMinutes(2),
      standardMinutes(17), standardMinutes(30), standardMinutes(
58), standardHours(2),
      standardHours(5), standardHours(8), standardHours(17), sta
ndardDays(1),
      standardDays(4))
    println(mean(durations).getStandardHours)
```

# 使用场景

`NumberLike` 类型类是一个非常好的例子，但 Scala 已经有 `Numeric` 了。 对于集合的类型参数 `T` ，只要存在一个可用的 `Numeric[T]` ，就可以在该集合上调用 `sum` 、 `product` 这样的方法。 标准库中另一个使用比较多的类型类是 `Ordering` ，可以为自定义类型提供一个隐式排序，用在 Scala 集合的 `sort` 方法。

标准库中还有更多这样的类型类，不过，Scala 开发者并不需要与它们中的每一个都打交道。

第三方库中一个非常常见的用例是对象序列化和反序列化，尤其是 JSON 对象。 使一个类成为某个格式器类型类的成员，就可以自定义类的序列化方式，序列化成 JSON、XML 或者是任何新的格式。

Scala 类型和数据库驱动支持的类型之间的映射，通常也是通过类型类获得自定义和可扩展性的。

# 总结

一旦开始用 Scala 来做些正式的工作，不可避免的会遇到类型类。 希望读者在读完这一章后，能够利用好这一强大技术。

Scala 类型类使得在开发 Scala 应用时，一方面可以有无限可追加的扩展， 另一方面又可以保留尽可能多的具体类型信息。

和其他语言应对这种问题的方法想比，Scala 给予了开发者完全的控制权，因为类型类的实现可以被轻易的重写，而且在全局命名空间里不可用。

你将看到这种技术在编写由其他人使用的库时尤其有用，在应用程序代码中，为了减少模块之间的耦合，类型类也是有用武之地的。

# 路径依赖类型

上一章介绍了类型类的概念，这种模式使设计出来的程序既拥抱扩展性，又不放弃具体的类型信息。 这一章，我们还将继续探究 Scala 的类型系统，讲讲另一个特性， 这个特性可以将 Scala 与其他主流编程语言区分开：依赖类型，特别是，路径依赖的类型和依赖方法类型。

一个广泛用于反对静态类型的论点是 “the compiler is just in the way”， 最终得到的都是数据，为什么还要建立一个复杂的类型层次结构？

到最后，静态类型的唯一目的就是，让“超级智能”的编译器来定期“羞辱”编程人员，以此来预防程序的 bug， 在事情变得糟糕之前，保证你做出正确的选择。

路径依赖类型是一种强大的工具，它把只有在运行期才知道的逻辑放在了类型里，编译器可以利用这一点减少甚至防止 bug 的引入。

有时候，意外的引入路径依赖类型可能会导致难堪的局面，尤其是当你从来没有听说过它。 因此，了解和熟悉它绝对是个好主意，不管以后要不要用。

## 问题

先从一个问题开始，这个问题可以由路径依赖类型帮我们解决： 在同人小说中，经常会发生一些骇人听闻的事情。 比如说，两个主角去约会，即使这样的情景有多么的不合常理，甚至还有穿越的同人小说，两个来自不同系列的角色互相约会。

不过，好的同人小说写手对此是不屑一顾的。肯定有什么模式来阻止这样的错误做法。 下面是这种领域模型的初版：

```
    object Franchise {
       case class Character(name: String)
     }
    class Franchise(name: String) {
      import Franchise.Character
      def createFanFiction(
        lovestruck: Character,
        objectOfDesire: Character): (Character, Character) = (lo
vestruck, objectOfDesire)
    }
```

角色用 `Character` 样例类表示， `Franchise` 类有一个方法，这个方法用来创建有关两个角色的小说。 下面代码创建了两个系列和一些角色：

```
    val starTrek = new Franchise("Star Trek")
    val starWars = new Franchise("Star Wars")

    val quark = Franchise.Character("Quark")
    val jadzia = Franchise.Character("Jadzia Dax")

    val luke = Franchise.Character("Luke Skywalker")
    val yoda = Franchise.Character("Yoda")
```

不幸的是，这一刻，我们无法阻止不好的事情发生：

```
    starTrek.createFanFiction(lovestruck = jadzia, objectOfDesir
e = luke)
```

多么恐怖的事情！某个人创建了一段同人小说，婕琪戴克斯和天行者卢克竟然在约会！我们不应该容忍这样的事情。

> 婕琪戴克斯：星际迷航中的角色：http://en.wikipedia.org/wiki/Jadzia_Dax 天行者卢克：星球大战中的角色：http://en.wikipedia.org/wiki/Luke_Skywalker

你的第一直觉可能是，在运行期做一些检查，保证约会的两个角色来自同一个特许商。 比如说：

```
object Franchise {
  case class Character(name: String, franchise: Franchise)
}
class Franchise(name: String) {
  import Franchise.Character
  def createFanFiction(
      lovestruck: Character,
      objectOfDesire: Character): (Character, Character) = {
    require(lovestruck.franchise == objectOfDesire.franchise
)
    (lovestruck, objectOfDesire)
  }
}
```

现在，每个角色都有一个指向所属发行商的引用，试图创建包含不同系列角色的小说会引发 `IllegalArgumentException` 异常。

# 路径依赖类型

这挺好，不是吗？毕竟这是被灌输多年的行为方式：快速失败。 然而，有了 Scala，我们能做的更好。 有一种可以更快速失败的方法，不是在运行期，而是在编译期。 为了实现它，我们需要将 `Character` 和它的 `Franchise` 之间的联系编码在类型层面上。

Scala 嵌套类型 工作的方式允许我们这样做。 一个嵌套类型被绑定在一个外层类型的实例上，而不是外层类型本身。 这意味着，如果将内部类型的一个实例用在包含它的外部类型实例外面，会出现编译错误：

```
class A {
  class B
  var b: Option[B] = None
}
val a1 = new A
val a2 = new A
val b1 = new a1.B
val b2 = new a2.B
a1.b = Some(b1)
a2.b = Some(b1) // does not compile
```

不能简单的将绑定在 `a2` 上的类型 `B` 的实例赋值给 `a1` 上的字段：前者的类型是 `a2.B` ，后者的类型是 `a1.B` 。中间的点语法代表类型的路径，这个路径通往其他类型的具体实例。因此命名为路径依赖类型。

下面的代码运用了这一技术：

```
    class Franchise(name: String) {
      case class Character(name: String)
      def createFanFictionWith(
        lovestruck: Character,
        objectOfDesire: Character): (Character, Character) = (lo
vestruck, objectOfDesire)
    }
```

这样，类型 `Character` 嵌套在 `Franchise` 里，它依赖于一个特定的 `Franchise` 实例。

重新创建几个角色和发行商：

```
val starTrek = new Franchise("Star Trek")
val starWars = new Franchise("Star Wars")

val quark = starTrek.Character("Quark")
val jadzia = starTrek.Character("Jadzia Dax")

val luke = starWars.Character("Luke Skywalker")
val yoda = starWars.Character("Yoda")
```

把角色放在一起构成小说：

```
    starTrek.createFanFictionWith(lovestruck = quark, objectOfDesire = jadzia)
    starWars.createFanFictionWith(lovestruck = luke, objectOfDesire = yoda)
```

顺利编译！接下来，试着去把 `jadzia` 和 `luke` 放在一起：

```
    starTrek.createFanFictionWith(lovestruck = jadzia, objectOfDesire = luke)
```

不应该的事情就会编译失败！编译器抱怨类型不匹配：

```
    found   : starWars.Character
    required: starTrek.Character
                   starTrek.createFanFictionWith(lovestruck = jadzia, objectOfDesire = luke)
```

即使这个方法不是在 `Franchise` 中定义的，这项技术同样可用。 这种情况下，可以使用依赖方法类型，一个参数的类型信息依赖于前面的参数。

```
    def createFanFiction(f: Franchise)(lovestruck: f.Character, objectOfDesire: f.Character) =
      (lovestruck, objectOfDesire)
```

可以看到， `lovestruck` 和 `objectOfDesire` 参数的类型依赖于传递给该方法的 `Franchise` 实例。 不过请注意：被依赖的实例只能在一个单独的参数列表里。

# 抽象类型成员

依赖方法类型通常和抽象类型成员一起使用。 假设我们在开发一个键值存储，只支持读取和存放操作，但是类型安全的。 下面是一个简化的实现：

```
object AwesomeDB {
  abstract class Key(name: String) {
    type Value
  }
}
import AwesomeDB.Key
class AwesomeDB {
  import collection.mutable.Map
  val data = Map.empty[Key, Any]
  def get(key: Key): Option[key.Value] = data.get(key).asInstanceOf[Option[key.Value]]
  def set(key: Key)(value: key.Value): Unit = data.update(key, value)
}
```

我们定义了一个含有抽象类型成员 `Value` 的类 `Key`。 `AwesomeDB` 中的方法可以引用这个抽象类型，即使不知道也不关心它到底是个什么表现形式。

定义一些想使用的具体的键：

```
trait IntValued extends Key {
 type Value = Int
}
trait StringValued extends Key {
  type Value = String
}
object Keys {
  val foo = new Key("foo") with IntValued
  val bar = new Key("bar") with StringValued
}
```

之后，就可以存放键值对了：

```
val dataStore = new AwesomeDB
dataStore.set(Keys.foo)(23)
val i: Option[Int] = dataStore.get(Keys.foo)
dataStore.set(Keys.foo)("23") // does not compile
```

# 实践中的路径依赖类型

在典型的 Scala 代码中，路径依赖类型并不是那么无处不在，但它确实是有很大的实践价值的，除了给同人小说建模之外。

最普遍的用法是和 **cake pattern** 一起使用，cake pattern 是一种组件组合和依赖管理的技术。 冠以这一点，可以参考 Debasish Ghosh 的 文章 。

把一些只有在运行期才知道的信息编码到类型里，比如说：异构列表、自然数的类型级别表示，以及在类型中携带大小的集合， 路径依赖类型和依赖方法类型有着至关重要的角色。 Miles Sabin 正在 Shapeless 中探索 Scala 类型系统的极限。

# 译者结语

到这里，有关 Scala 的知识已经讲的差不多了。 原博文还有两篇用来讲述 Akka Actor，但个人觉得放在新手指南里并不合适。 不是说 actor 不重要，毕竟不是核心库的一部分，对于理解 scala 这门语言用处不大。

所以，就到这里结束吧，希望读者能喜欢上 Scala。 另外，欢迎对此译文的任何建议和批评。